

*Learning Linux with
Raspberry Pi*



树莓派开始， 玩转Linux

Vamei 周昕梓 著
雷雨田 插画

从硬核的硬件hacking，到神秘的操作系统原理，都在这本易懂的书。



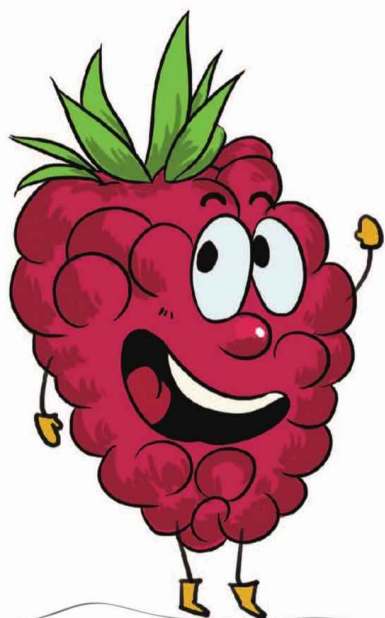
中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview[®]
www.broadview.com.cn

*Learning Linux with
Raspberry Pi*



树莓派开始， 玩转Linux

Vamei 周昕梓 著
雷雨田 插画

从硬核的硬件hacking，到神秘的操作系统原理，都在这本易懂的书中。



 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

张腾飞，笔名Vamei，中国科学技术大学学士，南洋理工大学博士，著有《从Python开始学编程》。技术博客博主，博客访问量超过800万次，现从事智能硬件的开发和创业。

周昕梓，本科毕业于南洋理工大学计算机科学与工程学院，现从事Hyperledger区块链相关软件开发工作。



作者豆瓣



树莓派开始， 玩转Linux

Vamei 周昕梓 著
雷雨田 插画

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内容简介

本书以树莓派为基础工具，讲解Linux操作系统。树莓派是近年来流行的微型电脑，能用于各种有趣的硬件开发。树莓派中安装了Linux系统，可以充当操作系统的学习平台。本书按照“树莓派背景——树莓派使用——Linux使用——操作系统原理——实操项目”的顺序展开。读者不仅能体验到玩树莓派的乐趣，而且能全面了解操作系统的核心概念和原理。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

树莓派开始，玩转Linux/Vamei，周昕梓著.—北京：电子工业出版社，2018.7

ISBN 978-7-121-34266-0

I. ①树... II. ①V... ②周... III. ①Linux操作系统—程序设计
IV. ①TP316.85

中国版本图书馆CIP数据核字(2018)第109364号

策划编辑：安 娜

责任编辑：汪达文

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×980 1/16 印张：22.25 字数：498千字

版 次：2018年7月第1版

印 次：2018年7月第1次印刷

印 数：2500册 定价：69.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。
若书店售缺，请与本社发行部联系，联系及邮购电话：（010）
88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至
dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序

2016年，Vamei同学的第一本书《从Python开始学编程》就给我留下了深刻的印象，我感觉写书对他来说是一个很快乐的过程，所以写出来的书读起来也令人感到轻松愉快。真是书如其人啊！

当Vamei同学把现在这本书给我看时，我会心一笑，正合我意。不仅因为他的书读起来很棒，而且因为我们正好也经常玩树莓派，主要是Raspberry Pi Hacking。我们把树莓派打造成黑客测试便携式工具，也把树莓派打造成自由上网路由器。Vamei这本书以树莓派为主线，介绍了Linux的相关知识与应用场景（包括现在很火热的区块链生态里的挖矿），由于树莓派的存在，这些应用场景更清晰、更形象。对于新手来说，这是本难得的入门好书；对于老手来说，这本书读起来充满了乐趣与思想碰撞。强烈推荐！

对了，写这个推荐序时，我也是轻松愉快的。好知识，就是有这样的感染力。

余弦

一名喜欢吃小龙虾的黑客

前言

我是抱着玩的心态开始用电脑的。自从家里有了电脑之后，我就想方设法抓起鼠标和键盘打一会儿游戏。《金庸群侠传》《仙剑》《星际》《盟军敢死队》，这些老游戏都玩了个遍。父母担心我沉迷游戏，一度没收了我的鼠标和键盘。总之，当时的电脑只是个娱乐平台。

那个时候已经在提“20世纪是计算机的世纪”。好莱坞电影开始把黑客塑造成孤胆侠客。我经常抱着《电脑报》或《大众软件》，幻想着自己成为一名侠客一般的计算机高手。但对于一个内地小城的孩子来说，深入接触计算机技术的机会很有限。我曾经很认真地找了一套计算机等级考试的书看，把二进制运算、SQL命令、QBasic语法都背得滚瓜烂熟，却因为装不好编译环境，最终没能写出一个可以使用的软件。读那些顶级黑客的传记，讲他们从小如何如何编程，一直很好奇他们是如何度过环境搭建这个难关的。后来发现，这些人都有机会接触一些编程高手，因此在他们眼里，这根本不是太大的问题。

上大学时，我选择了物理专业。物理专业做数值模拟和数据处理，C语言和Fortran语言编程也是必修课。有了大学里的资源，编程环境的搭建变成了小菜一碟。只是自己的电脑太过老迈，动不动就要死机。当朋友们呼啸着打Dota时，我却在为Word触发的蓝屏头痛。相熟的朋友看不下去，扔给我一张光盘，要我重装Ubuntu系统。Ubuntu是当时最流行的一个Linux版本。死马当活马医，我安装了光盘上的Ubuntu。系统装好了，电脑死机的次数大为减少。不过Linux下的图像化界面确实和Windows有差距，办公软件也比不上Office。我戚戚然地把Linux当作低成本的二等方案。但无论如何，当时正值我做“大学生研究计划”，运行稳定的Linux还是救我于水火。事后请朋友吃饭，问朋友哪里来的光盘，才晓得Ubuntu的安装光盘可以免费领取。

更让我刮目相看的是Linux下的软件分发。那个时代还没有苹果App Store这样的东西。所谓的在线软件分发，就是上网下载exe安装包。用了Ubuntu之后，我需要的软件基本都可以在软件源中找到。在

终端输入一行命令，编译环境就搭建好了。不用担心病毒，而且大部分情况下也不需要付费。再加上学校里有Ubuntu镜像，下载一个软件往往只需要几秒钟。于是，探索Linux下的软件成了我的一大业余爱好，我渐渐习惯了用ImageMagick来做图片处理，用FFmpeg来转换视频，用Wget来做网络下载。这些基于命令行的应用软件，再搭配bash的批处理功能，往往能实现强大的复合功能。

我也越来越享受Linux系统提供的编程环境。在写C语言和Fortran语言作业时，我就开始用vim编写自己的作业，用GCC和GFortran来编译，再用GDB来调试。这个过程要比Windows下的IDE麻烦。但当接触其他语言时，相同的工具可以复用，不用每一次都花费大量时间来熟悉全新的IDE。后来在Linux下学习Python语言时，很容易就可以上手。如果说编程是去游乐园，那么Linux就是为入园玩耍提供了直通车。想起小时候为编译环境苦恼的自己，真想穿越时空送去一张Ubuntu的安装盘。

我觉得对于一个电脑爱好者来说，Linux最美的地方就是开放。Linux的开放可以分为多个层面。软件层面是开放的，用户可以免费使用。文档也是开放的，你可以在终端下用man命令方便地查询。操作系统是开放的，你可以自由地调整系统，也可以深入了解其原理。代码上也是开放的，你随时可以看到世界上顶级程序员写的源代码。在Linux系统下，“实现”和“如何实现”是合二为一的。吃鱼的同时，钓鱼的本事也可以学到。因此，Linux提供了一个绝佳的学习平台。

后来，太太送给我一部树莓派作为生日礼物。我惊喜地发现，树莓派使用的操作系统正是Linux。更棒的是，树莓派的底层硬件也很开放。它可以方便地通过有线或无线的方式和硬件外设进行连接。它对使用方式没有太多限制。于是，在后来的智能硬件创业项目里，我总是在研发版本中使用树莓派。无论是作为硬件的树莓派，还是作为软件的Linux，都遵循了相同的规律：开放战胜了封闭。知识的共享带来了更加活跃的创造力，也给社会带来了协同合作的机会。

几年前，我读到印度的一个公益项目。这个项目募集旧电脑，在电脑上安装Linux系统，再发放给贫困地区的儿童使用。这个项目不仅给孩子们带来了欢乐，还改变了他们的命运。当树莓派发布的新闻出来时，我想到的就是这款微型电脑的社会意义。后来读到树莓派之父

厄普顿发明这台小电脑的初衷，果然也是教育。我由此确信，有很多人和我抱着相同的见解。

如今，“科技取代人类”的言论甚嚣尘上，很多人对技术霸权顶礼膜拜，对人类的未来充满绝望。其实，科技本身是中性的。科技可以取代人们的工作，也可以帮助人们更好地就业。像树莓派和Linux这样的技术，尊重了用户本身的创造力。它们用一种开放协作的态度，提高了社会的温度。我也一直抱着这样的理念，坚持在博客上分享自己的所知。我还记得自己在探索计算机时无路可循的尴尬。即使是出于简单的同理心，我也希望自己的分享能帮助任何一个在门槛上抓耳挠腮的学习者。

借着这股心劲，我克服了写作困难，全身心投入到本书的写作中。我希望这本书能以树莓派硬件为平台，全面讲解Linux原理。全靠昕梓的通力合作，我才能顺利完成这个野心勃勃的目标。杜鹃、陈思为帮我审读了全书，提出了大量的修改意见，让书稿变得真正可读。安娜会在关键的时候给我们提供任何所需的帮助，全程引导了写作过程。最后，这本书还要感谢上海地铁11号线。全靠这班地铁上的空座位，我才能坐着写出大部分文字。

在设计本书内容时，昕梓和我决定尊重读者，不避讳艰深的内容。毕竟，树莓派本身只是一个入口。这个入口的背后有着丰富的操作系统知识。无论是编程，还是深入理解计算机，一定深度的操作系统知识都不可或缺。我们会从树莓派的基本使用讲起，一直深入操作系统原理本身。在第5部分，我们还加入了基于树莓派的实践项目，希望能抛砖引玉，激发用户的创造力。当然，篇幅所限，也不得不舍弃一些细节，但我相信，只要体验到边玩边学电脑的乐趣，那么其他技术的掌握也都可以沿着相同的轨迹重复进行。

那样的话，这本书就没有遗憾了。

Vamei

2018.2.25

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

◎ **提交勘误**：您对书中内容的修改意见可在[提交勘误](#)处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

◎ **交流互动**：在页面下方[读者评论](#)处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34266>



目 录

内容简介

推荐序

前言

第1部分 怎样的树莓派

第1章 树莓派的诞生

第2章 树莓派的心脏

第3章 树莓派的大脑

第2部分 使用树莓派

第4章 开始使用树莓派

4.1 解剖树莓派

4.2 操作系统的安装与启动

4.3 图形化界面

4.4 Scratch

4.5 Kturtle

第5章 贝壳里的树莓派

5.1 初试Shell

5.2 用命令了解树莓派

5.3 什么是Shell

5.4 Shell的选择

5.5 命令的选项和参数

5.6 如何了解一个陌生的命令

5.7 Shell小窍门

第6章 好编辑

6.1 图形化的文本编辑器

6.2 使用nano

6.3 语法高亮

6.4 文件基础操作

第7章 更好的树莓派

7.1 常见初始化配置

7.2 软件升级与安装

第8章 漂洋过海连接你

8.1 局域网SSH登录

8.2 Bonjour

8.3 互联网SSH登录

8.4 文件传输

第9章 时间的故事

9.1 NTP服务

9.2 时区设置

9.3 实时时钟

9.4 date的用法

第10章 规划小能手

10.1 用cron规划任务

10.2 用cron开机启动

10.3 用/etc/init.d实现开机启动

10.4 避免使用/etc/rc.local

10.5 Shell中的定时功能

第11章 GPIO的触手

11.1 GPIO简介

11.2 控制LED灯

11.3 两个树莓派之间的GPIO

11.4 UART编程

11.5 用UART连接PC

11.6 用UART登录树莓派

第12章 玩转蓝牙

12.1 蓝牙介绍

12.2 BLE介绍

12.3 Bluez

12.4 了解树莓派上的蓝牙

12.5 树莓派作为BLE外设

12.6 树莓派作为BLE中心设备

12.7 树莓派作为Beacon

第13章 你是我的眼

13.1 摄像头的安装与设置

13.2 摄像头的基本使用

13.3 用VLC做网络摄像头

13.4 用Motion做动作捕捉

第3部分 进入Linux

第14章 Linux的真身

14.1 什么是内核

14.2 什么是GNU软件

14.3 Linux的发行版

第15章 你好，文件

15.1 路径与文件

15.2 目录

15.3 硬链接

15.4 软链接

15.5 文件操作

15.6 文件搜索

第16章 从程序到进程

16.1 指令

16.2 C程序

16.3 程序编译

16.4 看一眼进程

第17章 万物皆是文本流

17.1 文本流

17.2 标准输入、标准输出、标准错误

17.3 重新定向

17.4 管道

17.5 文本相关命令

第18章 我的地盘我做主

18.1 我是谁

18.2 root和用户创建

18.3 用户信息文件

18.4 文件权限

18.5 文件权限管理

第19章 会编程的bash（上）

19.1 变量

19.2 数学运算

19.3 返回代码

19.4 bash脚本

19.5 函数

19.6 跨脚本调用

第20章 会编程的bash（下）

20.1 逻辑判断

20.2 选择结构

20.3 循环结构

20.4 bash与C语言

第21章 完整架构

21.1 内核模式与系统调用

21.2 库函数

21.3 Shell

21.4 用户程序

第22章 函数调用与进程空间

22.1 函数调用

22.2 跳转

22.3 栈与情境切换

22.4 本地变量

22.5 全局变量和堆

第23章 穿越时空的信号

23.1 按键信号

23.2 kill命令

23.3 信号机制

23.4 信号处理

23.5 C程序中的信号

第4部分 深入Linux

第24章 进程的生与死

24.1 从init到进程树

24.2 fork系统调用

24.3 资源的fork

24.4 最小权限原则

24.5 进程的终结

第25章 进程间的悄悄话

25.1 管道

25.2 管道的创建

25.3 其他IPC方式

第26章 多任务与同步

26.1 并发与分时

26.2 多线程

26.3 竞态条件

26.4 多线程同步

第27章 进程调度

27.1 进程状态

27.2 进程的优先级

27.3 $O(n)$ 和 $O(1)$ 调度器

27.4 完全公平调度器

第28章 内存的一页故事

28.1 内存

28.2 虚拟内存

28.3 内存分页

28.4 多级分页表

第29章 仓库大管家

29.1 外部存储设备

29.2 外部存储器的挂载

29.3 ext文件系统

29.4 FAT文件系统

29.5 文件描述符

第30章 鸟瞰文件树

30.1 /boot和树莓派启动

30.2 应用程序相关

30.3 /etc与配置

30.4 系统信息与设备

30.5 其他目录

第31章 分级存储

31.1 CPU缓存

31.2 页交换

31.3 交换空间

31.4 外存的缓存与缓冲

第32章 遍阅网络协议

32.1 通信与互联网协议

32.2 协议分层

第33章 树莓派网络诊断

33.1 基础工具

33.2 网络层

33.3 路由

33.4 网络监听

33.5 域名解析

第5部分 树莓派小应用

第34章 树莓派平板电脑

34.1 平板电脑

34.2 硬件介绍

34.3 硬件的安装

34.4 配置操作系统

第35章 天气助手

35.1 读取互联网API

35.2 发送邮件

第36章 架设博客

36.1 安装服务器软件

36.2 安装Typecho

36.3 让别人可以访问你的网站

第37章 离线下载

37.1 安装下载工具Aria2

37.2 Aria2的使用

37.3 远程使用Aria2

37.4 安装图形化下载管理工具

第38章 访客登记系统

38.1 编写命令行小程序

38.2 尝试Tkinter

38.3 制作访客登记系统

38.4 访客名片和访客拍照

第39章 节能照明系统

39.1 传感器

39.2 读取传感器数据

39.3 控制照明电路

第40章 树莓派挖矿

40.1 比特币钱包

40.2 在树莓派上挖矿

40.3 区块链存储服务

第41章 高性能计算

41.1 Spark

41.2 树莓派与Spark

41.3 单机版 π 计算

41.4 树莓派集群

第42章 蓝牙即时通信

42.1 树莓派与蓝牙

42.2 蓝牙服务端

42.3 蓝牙客户端程序

42.4 服务端和客户端通信

42.5 实现文字聊天功能

42.6 数据加密传输

第43章 制作一个Shell

43.1 配置项目

43.2 输入输出设置

43.3 初步的Shell

43.4 文字颜色与其他配置

43.5 部分Shell 功能

43.6 Shell主程序

第44章 人工智能

44.1 树莓派的准备

44.2 YOLO识别

44.3 图形化显示结果

附录A 字符编码

附录B Linux命令速查

附录C C语言语法摘要

附录D Makefile基础

附录E gbd调试C程序

附录F 参考书目及简介

后记

第1部分 怎样的树莓派

树莓派是近年来诞生的一款微型电脑。由于体积小、功耗低，树莓派广泛应用于硬件创新领域，比如机器人、无人机、物联网和智慧工业。这一部分将介绍树莓派的相关背景：树莓派是如何诞生的，与树莓派相关的软硬件又分别有怎样的传奇故事。希望通过这一部分的介绍，能让你熟悉这款好玩的电脑，并认同它所秉承的开放创新理念。

第1章 树莓派的诞生

2006年，剑桥大学年轻的助教埃本·厄普顿（Eben Upton）（如图1-1所示）在为新入学的本科生头痛。



图1-1 埃本·厄普顿

无疑，那些能进入剑桥大学的新生都有聪明的脑瓜。他们拿着傲人的A-Level考试成绩进入计算机系。从成绩上看，这些野心勃勃的年轻人无可挑剔，可坐在电脑前，这些新生就露馅了。大多数人只会摆弄Word和Excel，水平好一些的，也只不过会做一两个简单的网页。新生们的计算机水平让厄普顿和他的同事们摇头不止。

曾经，玩计算机的都是一群极客。他们的考试成绩或许不是那么优异，也不一定能在考试中过五关斩六将进入剑桥。但这些极客都是从小玩着UNIX系统和编译器长大的。按理说，到了2006年，家用电脑早已普及，越来越多的学生乐于选择计算机作为专业，计算机水平应该越来越高超。然而，厄普顿看到的实际情况却是新生的计算机水平很糟糕。

当然，剑桥有能力把新生培养成合格的计算机专业毕业生，但像厄普顿这样的内行明白，高手的养成有赖于青少年时期的动手实践。

他自己就是个很好的例子。厄普顿成长于20世纪80年代。那个年代的英国人充满了动手精神。英国男人们以改装汽车和修冰箱为乐。厄普顿的父亲虽然是一位语言学教授，却也喜欢在业余时间带着自己的儿子们把引擎大卸八块，或者用继电器拼装起奇形怪状的家电。相同的手工精神也弥漫于计算机领域。计算机爱好者们不但对软件编程很熟练，对硬件调试也手到擒来。

在这种手工精神的鼓励下，市面上出现了很多为青少年设计的电脑，如Commodore 64和BBC Micro。Commodore 64是由Commodore公司推出的低端家用电脑，售价595美元。BBC Micro（如图1-2所示）是BBC电视台推出的教育电脑，单价200到300英镑。这些低端电脑性能一般，有时还会出不少bug。但它们售价便宜，让学校和普通家庭也可以轻松负担。由于母亲是学校老师，厄普顿本来可以免费使用学校的机房，但厄普顿的小伙伴们都有了自己的BBC Micro，而且一聚在一起就大聊各自的使用经验。不甘落后的厄普顿存够了200多英镑，给自己添置了一台BBC Micro。



图1-2 BBC Micro

这款电脑有些像中国早年流行的学习机，预装了很多游戏和教育软件。更令人惊讶的是，BBC Micro非常尊重孩子们使用电脑的自

由。它不但自带了编译环境，而且开放了大多数的设备接口。这意味着，电脑的功能全面地开放给了孩子们。如果喜欢，孩子们可以进行任何层面的编程，从而自由地发挥电脑的功能。对于厄普顿这样喜欢探索的孩子来说，BBC Micro提供了广阔的空间。

有一次，厄普顿想给自己的电脑增加一个鼠标，那时的鼠标可是新鲜出炉的“黑科技”。当然，新科技有很多不完善的地方，厄普顿买回来的鼠标就没有驱动。厄普顿的父亲帮他打电话到鼠标公司，结果对方的销售人员恶狠狠地回复：“如果你的儿子不会写驱动，那他就不该买鼠标。”

年少的厄普顿天真地相信了销售员的话，他决定自己写鼠标驱动。给硬件写驱动，大概是让成年人都会生畏的任务。幸好，BBC Micro开放的接口给鼠标驱动的开发提供了可能性。厄普顿用轮询的方式给自己的鼠标写了一个简单的驱动。当这个驱动运行时，这台简陋的BBC Micro就会变得异常缓慢，但总归可以看到鼠标的移动了。

相比于少年时的厄普顿，剑桥新生们能接触到性能高得多的电脑。这些电脑上配备的Windows系统，也比BBC Micro强大得多，但20世纪80年代的动手精神似乎忽然消失了。个人电脑成了很多家庭的工作和娱乐中心，花大价钱买高性能电脑的父母们，当然不想让自己的熊孩子把牛奶泼洒在键盘上。小孩子们再也不能像对待自己的BBC Micro那样，任意实验疯狂的想法。另一方面，新时代的电脑预装的都是Windows操作系统。Windows看似友好的图形化界面，把计算机真正的工作流程都隐藏在了幕后，让青少年们失去了进一步探索的动力。在Windows平台上，编程开发软件需要额外花钱购买。正因为如此，剑桥新生们反而没有20世纪80年代的厄普顿幸运。

少年时的情怀再次萌动，厄普顿想再造一台BBC Micro，让新时代的青少年可以尽情探索。他很快用各种电子元件在面包板上拼凑出一台粗糙无比的电脑，得意地展示给同事们。他的同事们都夸奖厄普顿“了不起”。可那些夸奖，听起来更像夸奖一个会打铁的现代人的老古董。毕竟，厄普顿的手工电脑性能太差。个人电脑尽管昂贵又没个性，却在性能上强大得多。没有哪个人会在家里用厄普顿的老古董。

厄普顿意识到，就算是简易电脑，还是要保持一定的性能。可是为了能让一般用户满意，成本就会迅速往上蹿。除非批量生产，否则简易电脑的成本根本无法降低到合理的水平。但厄普顿不知道自己的简易电脑能卖多少台，一千台？两千台？这样的订单数量，只会让供应商和制造工厂哂然一笑。

就在厄普顿想要放弃时，麻省理工学院传出消息，想要以Apple-II为蓝本，制作一款廉价的简易电脑。“我们可不能输。”这是剑桥计算机实验室的第一反应，所有人立刻想起厄普顿快要进棺材的手工电脑。名校的竞争精神再次复活了厄普顿的项目。一个以厄普顿为首的小圈子形成，他们用电子邮件积极交流制造廉价电脑的想法。为了方便指代厄普顿的电脑，一封邮件中使用了“树莓”这种水果的名字。树莓从此成为项目的代称。由于原型机上只支持Python编程语言，“树莓”后面又跟上了代表Python的“派”。树莓派（Raspberry Pi）就这样诞生了。厄普顿于2009年成立了慈善性质的“树莓派基金会”。这个基金会是管理运营树莓派的主要机构。树莓派的标志，如图1-3所示。

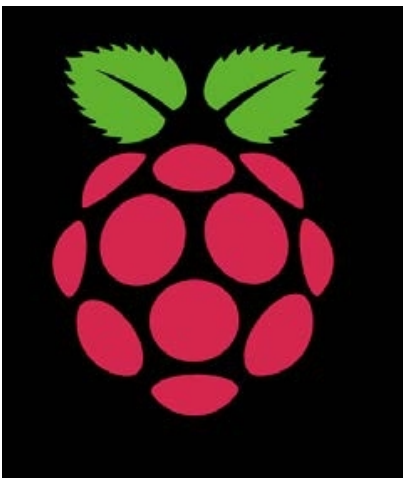


图1-3 树莓派的标志

2011年，BBC科技记者罗伊的一篇博客文章，让“树莓派”进入公众视野。很多极客开始关注树莓派的产品开发。但实验原型和正式产品还有很遥远的距离。成本是最大的挑战。厄普顿的销售定价是15英镑。但对于无法预估销售量的新产品，供应商不愿给出太多优惠。幸好，厄普顿新入职的工作岗位带来了机会。他供职的博通公司（Broadcom）正在为手机生产ARM处理器，其性能和成本正符合厄普

顿的预期。厄普顿决定使用ARM架构，从而解决了最关键的成本问题。

即便如此，树莓派的总成本还是难以控制在15英镑以下。很多同事劝厄普顿调高售价预期。但厄普顿不愿放弃，拼了命地想要压低每一分钱的成本。曾经的极客少年成了锱铢必较的“狡猾”商贩。他在市场上搜寻每一种型号的元器件，以便获得最优惠的价格。为了降低以太网接口的成本，他拜访了从原厂到代理到经销商的每一个环节，最终从一家欧洲经销商处获得了半价折扣。为了寻找合适的代工厂，他几乎走遍欧美和东亚，向经理们描述着自己的教育梦。冲着厄普顿的热忱，中国台湾的一家电路板厂商才以近乎赔钱的价格接下了最初的树莓派订单。

2012年2月，树莓派（如图1-4所示）终于解决了最后一个关键问题：把Linux操作系统导入到充当文件系统的SD卡上。这个信用卡大小的电脑，与这个星球上最流行的开源操作系统合体了。Linux平台的所有功能向树莓派开放。厄普顿终于实现了自己的目标。公众也对这款区区15英镑的电脑充满好奇，总是不停地访问树莓派官网，想要获知产品发售的消息，甚至造成网站不断地死机。在同年2月底，树莓派刚刚发售，订单就纷至沓来。厄普顿惊讶地发现，自己大着胆子准备的一万台树莓派很快就销售一空。他的问题变成了甜蜜的痛苦：如何生产更多的树莓派来满足市场需求。



图1-4 树莓派

从一开始，树莓派的影响就远远超出了教育领域。由于树莓派的小尺寸和低功耗，你可以把它当作很多移动平台的“大脑”。本来需要一台PC控制的机器人改用树莓派，体形一下就轻盈了许多。无人机控制同样是树莓派大显身手的地方。航天爱好者还把树莓派绑在高空气球上，以便能从几十公里的高空俯拍地球。英国宇航局甚至带了两块树莓派进空间站。树莓派成本低廉，立即成为智能家居和工业控制的重要组件。一些爱好者用树莓派来控制灯光和风扇，以便远程照顾自己的多肉植物。不少工厂用树莓派来执行关键作业，不但降低了成本，机器的运行也更加稳定。

厄普顿的简易电脑无意间填补了硬件开发的市场空白。那些支持厄普顿的供应商也因此获得了数百万的订单。但对于厄普顿来说，他最开心的事情，就是孩子们有了一款可以随便玩的电脑。直到今天，树莓派基金会依然致力于计算机教育项目。

第2章 树莓派的“心脏”

树莓派上最关键的元件，就是位于其中心位置的“心脏”——ARM处理器。这款处理器的诞生，也与BBC Micro这款启发了树莓派的电脑有关。1980年，“计算机”概念在欧美大热，成了电视和报纸上最常谈论的话题。BBC电视台趁机策划了一系列关于计算机的电视节目。导演遇到一个问题：怎么给没见过电脑的观众画饼。

此时，大洋彼岸的苹果公司已经推出了适合个人使用的微型电脑。Apple-II电脑在20世纪70年代末创造了销售神话，从而开发出个人电脑这个新市场。个人电脑在美国风靡，温吞的英国人的节奏却慢了一拍。对于英国人来说，计算机还是限于科研、国防、制造领域的高科技设备，和自己的生活会没有太大关系。美国舶来的个人电脑都售价不菲。英国人不愿用一年的茶钱来换一台用途不明的机器。在这种情况下，无论BBC主持人怎样能说会道，只能凭空想象的电视观众估计也熬不过5分钟。幸好，BBC是英国传媒业的龙头，不会轻易放弃。BBC公开招标一款廉价的微型计算机。

中标的是艾康电脑公司（Acorn Computer Company）。按现在的标准看，艾康电脑很不靠谱。这家公司才成立两年，规模也很小。艾康的起家业务是给赌博机生产控制器。这些控制器拥有运算和存储组件，勉强算是电脑。但控制器执行的是固定的程序，与多功能的个人电脑还有相当的距离。

艾康中标的主要原因是他们正好有一台符合BBC预期的原型机。于是，这款原型机被重新命名为BBC Micro，成为电视节目的指定用机。借着电视节目，BBC Micro成为英国最流行的个人电脑。但钱没能消除艾康的危机感。与市面上其他的个人电脑相比，BBC Micro的性能没有优势。为了在竞争中胜出，艾康公司想把强大的Intel处理器用在BBC Micro上。

处理器又被称为“中央处理器”或“CPU”，是计算机执行指令的中枢。所谓的指令，就是计算机的某个单元操作。我们在生活中经常下指令，比如要求别人“向左转”或“向右转”。同样，用户也可以向计算

机发出指令，比如要求计算机进行加减运算。无论多么复杂的动作，最终都会被分解为一系列的处理器指令来完成。因此，处理器的好坏直接决定了计算机的性能。

当时的Intel正风光无两。借着IBM电脑的大卖，Intel处理器（如图2-1所示）几乎占据了整个个人电脑市场。因此，Intel对于艾康这样的小客户提不起兴趣，不愿给出太大的折扣。由于BBC Micro的定位是廉价的教育型电脑，因此艾康最终放弃了Intel处理器，转而自行研发处理器。处理器的研发耗费巨大。艾康的工程师必须“事先非常仔细地考虑好所有的细节”，才能在苛刻的成本限制下实现处理器性能的提升。1985年，艾康公司给BBC Micro换上了性能优良的新型处理器。艾康公司也借此有了一个新产品——ARM处理器，如图2-2所示。



图2-1 Intel处理器

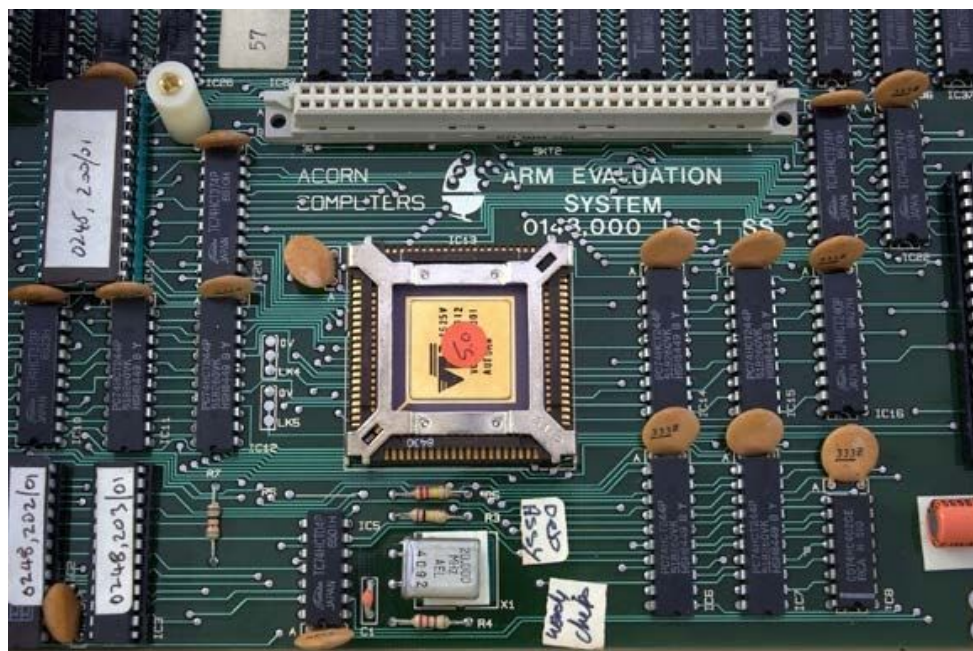


图2-2 BBC Micro中的ARM处理器

ARM是“Acorn RISC Machine”的简称，名字中的“RISC”，指的是ARM处理器对精简指令集的支持。这四个看起来干巴巴的字母，却是对Intel最直接的叫板。原因很简单，Intel采用的是完全相反的“CISC”。所谓的RISC，是指该类型的处理器只支持基本的汇编指令。“R”代表了“Reduced”，即“精简”。Intel支持的“CISC”，指的是复杂指令集。首字母“C”代表了“Complex”，即“复杂”。Intel处理器提供了比ARM处理器多得多的指令。和RISC相比，CISC处理器有很多高级功能，结构也相应复杂得多。从直觉上看，CISC处理器像是一辆超级跑车，让人趋之若鹜。

但两大阵营的对比并非那么简单。RISC支持的指令虽然基础，但总可以通过基础指令的组合来实现CISC处理器的功能。这意味着RISC处理器的汇编程序需要占用更多的空间，编译起来也比较耗时。但RISC处理器结构简单，制造成本低，运行起来也比较省电。其实在威尔森之前，大型服务器已经开始使用RISC处理器。这些大型电脑配备数目众多的处理器，就好像拥有大量汽车的出租车公司，更愿意选择经济型轿车。艾康的独到之处是把RISC引入了低成本的小型设备。

凭着ARM处理器，艾康守住了教育电脑市场。BBC Micro销量达到上百万台，直到1994年才彻底停产。但在更广阔的个人电脑市场上，Intel的CISC处理器才是赢家。毕竟，个人电脑逐渐成为家庭娱乐

和个人办公的中心。一台个人电脑往往会使用5到7年，而电脑上的软件也会越来越多、越来越耗费资源。为了应对漫长的使用期，用户当然希望自己手里的是一辆超级跑车。因此，Intel长期霸占个人电脑市场，只留给竞争对手一点边角料。艾康想扩大份额，只能靠个人电脑之外的应用场景。

艾康寄希望于苹果公司的新产品。1990年，艾康公司和苹果公司联合成立了ARM公司。ARM公司的设立充满实验性质。公司最初只有12个人，只能在一间谷仓里办公。这个小团队负责开发ARM处理器。苹果将ARM处理器用于牛顿掌上电脑（Apple Newton）。这款产品极具创意。其大屏显示和手写识别，直接启发了“商务通”等PDA（Personal Digital Assistant）产品，甚至影响了iPhone的设计，如图2-3所示。掌上电脑对性能的要求没有个人电脑那么高，但需要节约使用电池。低功耗的ARM处理器正适合。但“牛顿”是一款早熟的产品，因此没能获得商业上的成功。它售价太高，而关键性的手写功能又充满缺陷，造成该产品在商业市场上折戟。ARM的路似乎走到了尽头。

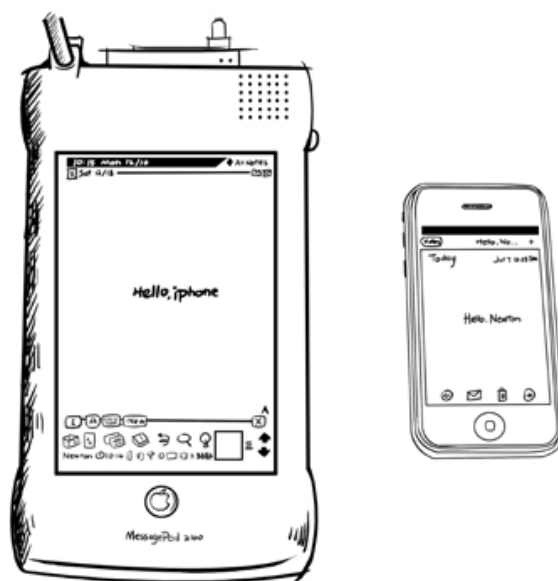


图2-3 Newton与iPhone

在这种绝望的状况下，ARM干脆彻底放弃了处理器的生产和销售。如果一家饭店既不做饭又不卖饭，估计第二天就要关张。幸好ARM公司是一家电子公司，还可以卖设计图纸。ARM当然不是自暴自弃地清仓甩卖。它收取一定的费用，把相关设计分享给有能力生产和销售的合作伙件。合作伙件生产出的每件ARM处理器，都要付给

ARM公司一定的授权费。通过这种授权知识产权模式，ARM省去了生产和销售环节的巨额成本。专注于上游的设计，这也让ARM公司快速地迭代开发。当然，这也是没办法的办法。Intel这样的霸主，包揽了从设计到销售的全链条，根本不用像ARM这样委曲求全。

ARM开放的合作框架，掀起了一场反抗Intel的暴动。很多电子元件厂商都想和Intel竞争处理器市场，但都忌惮Intel的强势，不敢轻易涉入。与ARM公司合作，成了“蜀汉联合，共抗曹魏”的理想策略。反过来，这些厂商上了船，也心甘情愿地为ARM处理器攻城略地。德州仪器公司（Texas Instrument）生产的ARM处理器，就被诺基亚用在6110手机上。这款手机在中国也曾红极一时，笔者就曾拿着老爸的6110使劲地折叠贪吃蛇。除了6110这样的明星产品，ARM处理器还收编了诸多细分领域。在低端领域，ARM处理器“够用就好”的原则正好可以控制成本。在专用设备方面，ARM开放的架构允许小型电子厂自由地定制，也广受欢迎。

就在ARM攒足粮草的关键时机，苹果终于发力助攻。乔布斯回归苹果，发布了革命性的iPhone。由于iPhone选用了ARM处理器，因此ARM的市场份额开始狂飙。事实上，Intel曾有机会拿下iPhone。在iPhone诞生之前，苹果就和Intel达成战略合作关系，并把Intel处理器应用于苹果电脑。苹果也有意委托Intel开发iPhone的处理器，只是Intel内部并不看好iPhone，担心收不回投资成本。

ARM的开放又一次战胜了Intel的封闭。随后，谷歌推出安卓操作系统，刺激出一众安卓手机厂商。寻求快速迭代的安卓厂商很自然地选用开放的ARM处理器。ARM在手机市场的狂飙让Intel人心不稳。苹果在平板电脑iPad上再次跳过Intel，使用了ARM处理器。业界议论纷纷，既然ARM处理器能满足平板电脑的性能需求，为什么不能用于Intel坐镇的高端个人电脑呢？Intel的霸主地位日渐动摇。

如今，ARM处理器的出货量已经远远超过Intel，并占据了90%以上的手机处理器市场。树莓派使用的是来自博通公司的ARM处理器，从而为ARM处理器探索出新的应用领域。同样承袭BBC Micro的衣钵，ARM和树莓派都为计算机领域开创了新的发展模式。

第3章 树莓派的大脑

如果说ARM处理器是树莓派的**心脏**，那么Linux操作系统就是树莓派的**大脑**。大多数树莓派上安装的都是Linux操作系统。树莓派官方推出的Raspian操作系统，也是Linux的一个发行版本。

Linux操作系统是林纳斯·托瓦兹（Linus Torvalds）（如图3-1所示）在1991年创造出来的。1991年，托瓦兹还是一名普通的大学生，刚刚买了一台3500美元的电脑。这对于任何一个芬兰家庭来说都是奢侈品。更何况，托瓦兹的父母没有太多闲钱来赞助儿子。托瓦兹把奖学金和零用钱加在一起，付了电脑三分之二的钱。剩下的三分之一，要在接下来的三年里分期支付。拿到电脑之后，托瓦兹连着几个月都耗在上面。不过，托瓦兹的母亲对此并没有太大意见，只是偶尔会提醒托瓦兹吃饭。倒是妹妹萨拉会在隔壁咆哮，逼着正在拨号上网的哥哥让出电话线。



图3-1 林纳斯·托瓦兹

由于父母早年离异，所以托瓦兹大部分时间都是跟着母亲生活的。他的外公是一位统计学教授，因此有一台工作用的Commodore电脑。这个品牌的电脑和BBC Micro一样，都曾在欧洲流行。不知是为了培养外孙，还是纯粹的偷懒，外公经常会口述程序，让托瓦兹敲入

电脑里。年幼的托瓦兹很快发现，这个其貌不扬的“盒子”并不介意用户是个儿童，只要输入程序，电脑就会根据指令工作，不多也不少。除了服兵役的将近一年时间，托瓦兹把大部分时间都花在电脑编程上。考入赫尔辛基大学时，托瓦兹已经有了丰富的编程经验。

托瓦兹写了一个终端模拟程序。通过这个程序，托瓦兹可以通过电话线连接学校机房的电脑，再通过机房的电脑在互联网上收发邮件。在20世纪90年代初，电子邮件还是少数“极客”才能玩得转的高科技，一般人甚至不知道电子邮件是什么。因此，当托瓦兹向妹妹展示终端模拟器时，萨拉一脸茫然，完全不知道哥哥在干什么。托瓦兹很难向妹妹解释清楚这个程序的厉害之处。这个程序是用汇编语言写的，可以直接和电脑硬件互动。换句话说，对于一台没有安装类似Windows这样操作系统的电脑，托瓦兹可以让它运行《魔兽争霸》。当然，托瓦兹实现的功能要比游戏简单得多。

托瓦兹的野心当然不止于此，他准备让自己的操作系统超越UNIX。UNIX是一个操作系统程序，比Windows年长了20岁。贝尔实验室的肯·汤普森（Ken Thompson）想在一台PDP-11型号的电脑上玩一款叫作《太空旅行》的游戏，就和同事丹尼斯·里奇（Dennis Ritchie）一起编写了UNIX操作系统，如图3-2所示。和之前的操作系统相比，UNIX非常简单。计算机的各项活动，无论是用户交互，还是编译程序，都组织成结构相似而在运行上相互独立的“进程”。进程之间可以通过文本形式相互通信，从而能协同工作。计算机上的数据，从程序文本，到配置信息，再到硬件接口，都存储成文件。UNIX与其说是一个程序，倒不如说是一套关于操作系统的哲学。肯·汤普森就好像计算机世界里的牛顿，把计算机可以实现的复杂活动分解成几条简单的物理定律。UNIX流行了将近半个世纪，并影响了非UNIX阵营的其他操作系统，如微软的MS-DOS和Windows。



图3-2 肯·汤普森和丹尼斯·里奇在一台PDP-11前

拥有贝尔实验室的AT&T（美国电信电报公司）当时有政府禁令在身，不能涉足软件业务。因此AT&T允许教育机构免费使用UNIX。因此，UNIX在大学里传播得很快。肯·汤普森的母校伯克利大学推出了一个更加好用的BSD（Berkeley Software Distribution）版本。因为这些计算机系的大学生用惯了UNIX，所以步入社会之后，也把UNIX推广到了IT公司。UNIX成为黄金万两的生意，并衍生出各种各样的商用版本。赫尔辛基大学也在刚刚购置的小型机上安装了UNIX，可以让十多个学生同时在线使用。托瓦兹就是这台电脑的常客之一，并很快喜欢上了UNIX。他不但花了一整个夏天去钻研操作系统的经典教材，还学会了用来开发UNIX程序的C语言。只可惜，UNIX对于家用并不免费，一个最便宜的UNIX系统也要数千美元，已经负债累累的托瓦兹可负担不起。

为了从成熟的UNIX体系下借力，托瓦兹把UNIX操作系统下常用的文本交互器bash嫁接到自己的终端模拟程序上。有了这个文本交互界面，家里的电脑就像学校里的UNIX一样好用。他很快又给自己的电脑安装了C语言编译器gcc。由于UNIX下的大部分应用程序都是用C编写的，所以托瓦兹可以在自己的操作系统中编译几乎所有的UNIX应用程序。托瓦兹意识到，自己的操作系统越来越完善。他又一次充满了创造者的骄傲。

1991年8月，托瓦兹在Minix新闻组上发帖：

各位Minix用户，大家好。我正在制作一个（免费）的操作系统（只是作为爱好，不会像gnu那样专业）。这个项目从4月份就启动了，并将要准备好了。我想听听大家的意见，特别是大家喜欢或不喜欢Minix的地方，因为我的操作系统将会和Minix有些像。我正在移植bash和gcc。这意味着在接下来的几个月里，我将获得一些实质性的成果.....此外，它没有用Minix的代码.....

那个时候，Minix是操作系统世界里的明星。编写Minix的是阿姆斯特丹自由大学的一位计算机教授安德鲁·塔能鲍姆（Andrew Tananbaum）。为了教学方便，他仿照UNIX编写了Minix这款操作系统，并开放源代码，以便学生更好地理解操作系统的原理。他编著的操作系统教材也非常畅销。托瓦兹就是借着那本700多页的教科书才摸清操作系统原理的。多年之后，托瓦兹在阿姆斯特丹自由大学演讲时，曾拿着那本书想获得塔能鲍姆的签名。很不巧的是塔能鲍姆正好不在。

Minix并不如UNIX成熟，但比起托瓦兹的操作系统还是强得多。Minix已经有不少拥趸者。还有不少高手给Minix编写补丁，大大提高了Minix的可用性。托瓦兹自己工作时，主要用的就是Minix。因此，托瓦兹在Minix新闻组里发布自己的操作系统，看起来就像是闯入瓷器店惹事的公牛。意外的是，托瓦兹在新闻组里获得了不少支持。发帖不久，就有Minix用户向托瓦兹反馈，说明自己想要的功能。有的用户还为托瓦兹建立了FTP服务器，用于上传正式发布的操作系统代码。Minix用户看起来有些薄情，但这应该归咎于塔能鲍姆。他有言在先，不希望人们拓展他的源代码。即使有热心用户编写了改进程序，塔能鲍姆也不会把这些改进加入正式发行的版本里。因此，人们只能编写非正式的补丁并私下交流。Minix的发展陷入停滞。

相反，托瓦兹采用了GPL协议（General Public License）。任何用户都可以自由地使用并修改GPL协议的代码，但基于此修改出的代码，也必须遵照GPL协议开放，供他人使用或修改。这个行动充满了理想主义的味道，意味着托瓦兹不能从自己编写的程序获得直接的经济利益。考虑到托瓦兹的父母都曾是学生运动领袖，他的父亲还是芬兰左翼的重要成员，有人疑心托瓦兹的做法来自于家庭的影响。但按照托瓦兹自己的解释，他用GPL的唯一原因就是懒。有了GPL协议，爱好者们可以毫无顾忌地贡献代码。他只要从中择优，加入正式版本中，就可以省掉自己开发的麻烦。这一“诡计”确实奏效。爱好者们不

但贡献了代码，还凑钱帮托瓦兹还了买电脑的欠债。他们还用托瓦兹的名字“Linus”命名这个操作系统为“Linux”。最后一个字母，按照UNIX的传统改成字母“x”。Linux系统的标志，如图3-3所示。



图3-3 Linux系统的标志：企鹅

圈内的很多人都不看好Linux。在Linux出生大约一年之后，UNIX之父汤普森和Minix之父塔能鲍姆公开批评Linux的实现方式。塔能鲍姆甚至说，如果托瓦兹是他班上的学生，那这个学生的成绩一定不及格。开源运动领袖艾里克·雷蒙（Eric Raymond）后来回忆，当他第一次接触Linux代码时，他有理由相信Linux最终会失败。显然，他们低估了社区的重要性。即便托瓦兹不是最天才的程序员，但社区爱好者的贡献能让任何天才程序员都跟不上Linux的速度。另一方面，托瓦兹在保持开源理想的同时，又有足够的实用精神。Linux采用了GPL协议，但托瓦兹并不鼓吹“自由软件就是好”的绝对论断。在他看来，无论哪一种力量，商业也好，非商业也好，只要能促进Linux的发展，就可以为Linux所用。在遇到问题时，托瓦兹也不会陷入“完美系统”的洁癖。他愿意接受一个不甚完美的方案，然后快速迭代，不断优化方案。同样采用GPL协议，但更富有理想主义的GNU项目也在内核开发上败给了Linux。

1995年，用于HTTP服务的Apache服务器发布。互联网服务商发现，可以把同样免费的Linux和Apache服务器结合在一起，廉价地搭建网站所需的服务器。此时的Linux已经疯狂进化了好几年，强健到完全

可以胜任网站服务器的工作。内容丰富的网页取代了电邮和新闻组，成为互联网的主流。基于这套技术，最早的一批互联网公司建立起来，如雅虎、亚马逊，以及中国的搜狐。“dot-com”热潮给Linux打了一剂强心针。在网络服务器市场上，Linux彻底打败微软的Windows NT，成为大多数互联网公司的选择。网景、甲骨文、IBM等公司开始支持Linux系统，甚至同意把自己的部分代码公开，贡献给开源运动。托瓦兹的照片因此登上了福布斯的封面，成为很多青少年的偶像。

来自芬兰的穷小子打败了一统天下的比尔·盖茨，这本来就是话题性十足的故事。更让人感到困惑的是，免费的Linux究竟怎么赚钱。记者们抢着给托瓦兹打电话，想要获得独家采访的机会。他们意外地发现，接电话的并非助手，而是这个传奇英雄本人。事实上，托瓦兹也从来没有私人助手。尽管Linux项目有数万的参与者，但这些参与者组织成了不同的项目。托瓦兹真正需要打交道的，只是几十个项目领导人。另一方面，尽管领导着人类历史上规模最大的软件合作项目，甚至坐拥Linux这个商标，但托瓦兹并不富有。1997年，托瓦兹带着妻子和刚出生的女儿迁居美国，他的账户只有几千美元的余额。在美国的第一个晚上，托瓦兹不得不和妻女挤在充气床垫上，而他的猫咪也只能睡在旅行用的笼子里。

不过，如果托瓦兹愿意，他完全可以凭自己的身份获得更好的生活。微软的史蒂夫·巴尔默对Linux极为警惕，而史蒂夫·乔布斯曾亲自邀请托瓦兹加盟苹果。红帽Linux和VA Linux这些提供Linux服务和支持的公司也成立起来，获得了令人瞩目的成功。托瓦兹接受了这些公司为表达感谢而赠送给他的期权，却不愿到其中任何一家任职。托瓦兹乐意看到Linux在商业上的突破。他只是在做个人选择时极为谨慎，免得自己因为商业利益而无法保持中立。

不过，生活总是给托瓦兹带来意外的惊喜。随着红帽Linux和VA Linux的上市，托瓦兹手里的股票价值一度高达2000万美元。但托瓦兹还是住在普通的房子里，把大部分时间花在维护Linux上。真正令托瓦兹骄傲的是，社会彻底改变了对像他这样的极客的看法。极客不再是20世纪七八十年代留着长胡子，穿着拖鞋整日躲在黑暗房间里的怪胎。相反，人们把他们看成技术先锋。大公司愿意出高薪聘用参与Linux核心项目的程序员。除了高超的技术，这些为开源社区做贡献的极客们还带来了一种已经改变了历史的软件开发方式。

如今的杂志封面上，托瓦兹的Linux已经被人工智能、手机、虚拟现实、物联网取代，但Linux并未退休，只是沉淀为技术世界不可或缺的基础设施。想想吧，在IBM的超级电脑、谷歌的安卓手机、虚拟现实和物联网的嵌入式设备上，都运行着Linux系统。另一方面，像树莓派这样配备了Linux的超小型电脑，可以自由地使用Linux孕育出的代码库，从而极大地扩展了设备的可用性。正因为如此，学习Linux成为玩转树莓派的关键。我们也将从树莓派开始，深入学习Linux。

第2部分 使用树莓派

听了那么多树莓派的故事，你一定想见识一下它的真面目。本书将从安装开始，逐步深入树莓派的使用。这一部分内容偏实用，旨在让你实际体验树莓派的功能。树莓派是一款特别适用于硬件互动的微型电脑，在这一部分会专门介绍树莓派这一方面的特长，如摄像头、GPIO接口和蓝牙模块。

第4章 开始使用树莓派

树莓派是一款信用卡大小的超小型电脑。它的长度为8.56cm，宽度为5.6cm，厚度只有2.1cm。树莓派把整个系统集成在一块电路板上的解决方案，被称为SoC（System on Chip）。SoC在手机等小型化设备中很常见，功耗也比较低。树莓派使用SoC的解决方案，正适合其超小型电脑的应用场景。

4.1 解剖树莓派

树莓派是一台功能完整的电脑。现代电脑都采用了冯·诺依曼体系。冯·诺依曼在1945年发表了一份报告，把计算机分为五大组件，如图4-1所示，树莓派也不例外。这五大组件分别如下。

1. 控制器

计算机的指挥部，管理计算机其他部分的工作，决定执行指令的顺序，控制不同部件之间的数据交流。

2. 运算器

顾名思义，这是计算机中进行运算的部件。除加、减、乘、除等算术运算外，还能进行与、或、非等逻辑运算。运算器与控制器一起构成了**中央处理器**（CPU，Central Processing Unit）。

3. 存储器

存储信息的部件。冯·诺依曼根据自己在曼哈顿工程中的经验，提出了存储器不但要记录数据，还要记录所要执行的程序。

4. 输入设备

向计算机输入信息的设备，如键盘、鼠标、摄像头等。

5. 输出设备

计算机向外输出信息的设备，如显示屏、打印机、音响等。

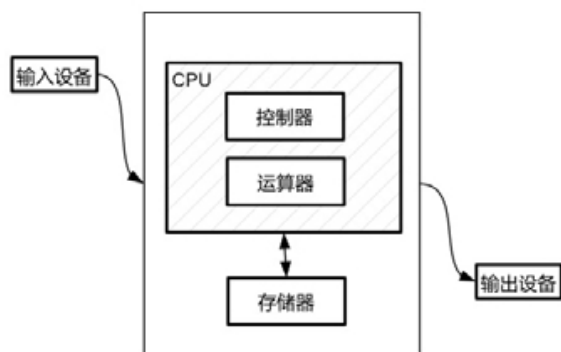


图4-1 冯·诺依曼体系

我们拿树莓派和冯·诺依曼体系做一个对比。来自博通公司的ARM CPU位于树莓派的正面，如图4-2所示。CPU中除了运算器、控制器和缓存，还有一块用于图形运算的GPU。内存位于树莓派的反面，提供了1GB的存储器空间，如图4-3所示。树莓派上并没有直接的输入输出设备，但预留了多种多样的接口。你可以通过这些接口来连接输入输出设备，例如用USB口连接键盘、鼠标，用HDMI口连接显示器。加上输入输出设备之后，树莓派就补齐了冯·诺依曼体系的五大组件。

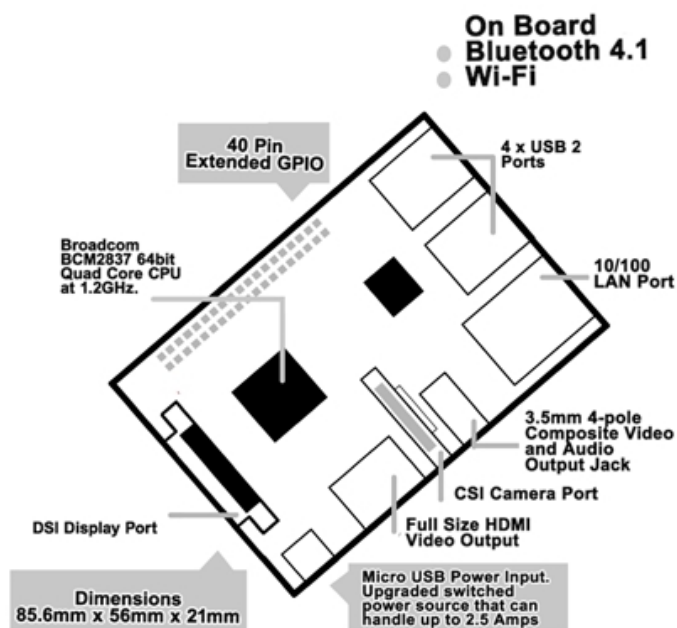


图4-2 树莓派的正面

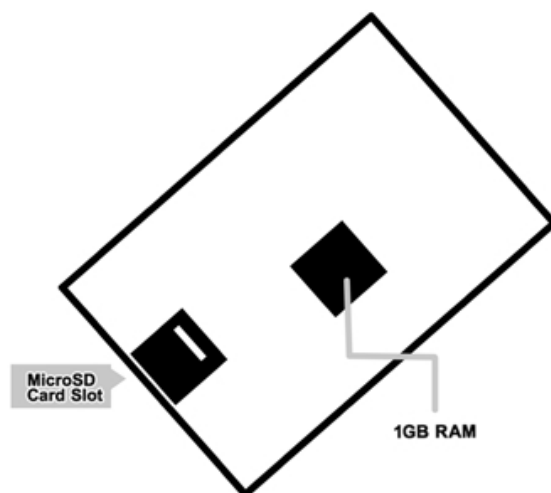


图4-3 树莓派的反面

为了启动设备，你还需要一个电压为5V的电源。这个电源通过Micro B USB的输出端和树莓派连接。树莓派官方售卖的电源插座可以直接插到家用的220V电压插座上，另一端的Micro B USB就可以插入树莓派。你也可以买一根USB转Micro B USB的连接线，把USB一端插入PC或其他提供电源的USB端口。一旦接上电，树莓派的电源指示灯就会亮起，系统自动启动。

此外，你还需要一张Micro SD卡来作为计算机的外部存储器。这张SD卡插入树莓派的卡槽中，就可以接入系统。冯·诺依曼并未区分内部存储器和外部存储器。但现代的计算机，内存和外存承担了不同的功能。一般家用PC除了内存，也会有磁盘或者固态硬盘这样的外部存储器。通常来说，内存容量较小，读写速度快，因此只用于存储与当前运行程序相关的信息。一旦断电，内存中的数据就会丢失。外存容量较大，外存选取的都是可以长期保存数据的介质，从而在断电期间也能保存数据。因此，需要长期保存的数据，如树莓派的操作系统，就需要保存在这张SD卡上。当树莓派开机时，会加载SD卡中保存的操作系统程序。用户产生的文件，也保存在这张SD卡上。如果你的树莓派硬件出现故障，只需把这张Micro SD卡插入其他树莓派中继续使用即可，而不用担心任何的数据丢失。

除了上面的核心组件外，树莓派还包括了其他接口。

- 40 PIN Extended GPIO：广义编程接口，常用于硬件控制。
- 10/100 LAN Port：用于有线连接的以太网口。

- CSI Camera Port：摄像头接口。
- 3.5mm 4-Pole Composite Video and Audio Output Jack：视频音频输出口。这个接口常用于音频输出。
- DSI Display Port：另一种显示接口。这个接口不常见。

在树莓派3 Model B中，还自带了Wi-Fi和蓝牙模块，可以方便地进行无线连接。在老版本的树莓派中，只能通过USB外接相关适配器，以实现无线连接。

4.2 操作系统的安装与启动

树莓派上最基础的软件就是操作系统。我们说过，树莓派的操作系统保存在它的外部存储器，也就是Micro SD卡上。由于树莓派一开机就需要找到操作系统，所以Micro SD卡上的操作系统程序必须提前写入。树莓派官方推荐使用8GB的SD卡，Raspbian操作系统本身占据的空间不到5GB，剩下的空间就可以由用户使用。不过，为了让空间足够充裕，建议使用更大空间的Micro SD卡。

我们需要一台可以读写Micro SD卡的电脑来烧录。很多笔记本电脑都自带SD卡插口。需要注意的是，这些SD卡插槽会比Micro SD卡尺寸大，所以需要有一个Micro SD卡转SD卡的卡套，才能插入插槽。即使没有SD插槽和转换卡套，一个USB接口的Micro SD卡读卡器也可以很便宜地买到。

本书的操作系统是树莓派官方推出的Raspbian。正如名字暗示的，Raspbian继承自Debian操作系统。Debian是Linux的一个发行版本。当你熟悉了Raspbian，就有了使用Linux系统的经验。官网提供了Raspbian的镜像文件，你可以到官网下载。由于Raspbian不时会更新版本，所以下载文件的名称也会有差异。本书后面把该镜像文件统称为raspbian.image。我们需要把这个镜像文件烧录到SD卡上。下面分别列出了多种烧录方式。

1.Windows电脑烧录

Windows系统有现成的图形化软件来完成上述镜像烧录工作，比如树莓派官网推荐的Win32 Disk Imager，如图4-4所示。

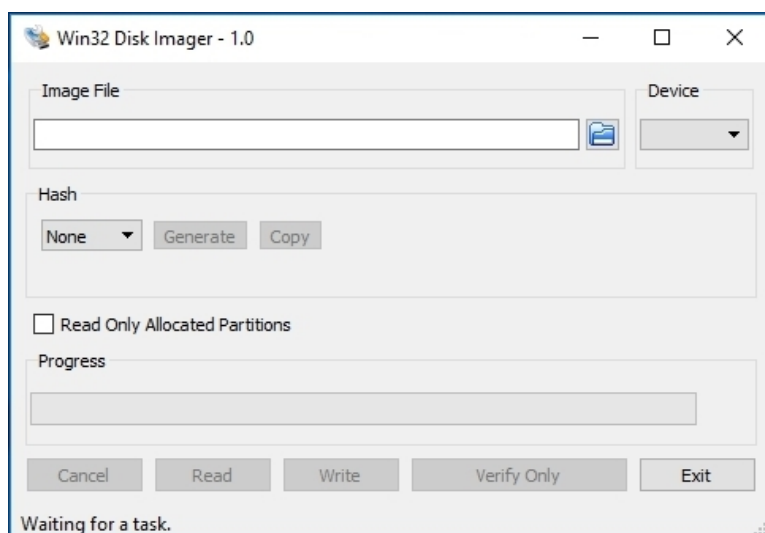


图4-4 Win32 Disk Imager界面

安装好Win32 Disk Imager后启动。在界面上的“Image File”栏选择镜像文件，在“Device”栏中选择Micro SD卡对应的盘符，然后单击“Write”按钮。待进度条完成后，Micro SD卡就烧录成功了。烧录完成后，把SD卡插入树莓派的卡槽中，为树莓派连通电源和显示器，就可以从屏幕上看到树莓派的启动画面了。

2. Mac OS X烧录

如果是在Mac OS X下，可以通过**终端**（Terminal）中的命令来进行烧录。首先，列出挂载的所有存储设备：

```
$diskutil list
```

从中找到对应Micro SD卡的设备，并记下它的路径，如/dev/disk3。然后，使用dd命令把镜像文件写入SD卡：

```
$sudo dd if=/dev/disk3 of=./raspian.image
```

3. Linux烧录

如果是在Linux系统下，那么可以用如下命令来找出Micro SD卡挂载的路径：

```
$sudo fdisk -l
```

Linux下的烧录和Mac OS X类似，可以使用dd命令，把镜像文件写入SD卡：

```
$sudo dd if=/dev/disk3 of=./raspian.image
```

4.NOOBS

NOOBS是一种更加简便的安装方式。NOOBS（New Out Of Box Software）是一个简便的操作系统安装程序。首先准备一个格式化为FAT格式的Micro SD卡，然后把解压缩后的NOOBS文件直接复制到SD卡中即可。随后，将此卡插入树莓派中，即可根据图形化界面提醒安装想要使用的操作系统。

4.3 图形化界面

开机完成后，就可以进入Raspbian的图形化桌面了。图形化桌面提供的主要功能都在上方的导航栏中，如图4-5所示。

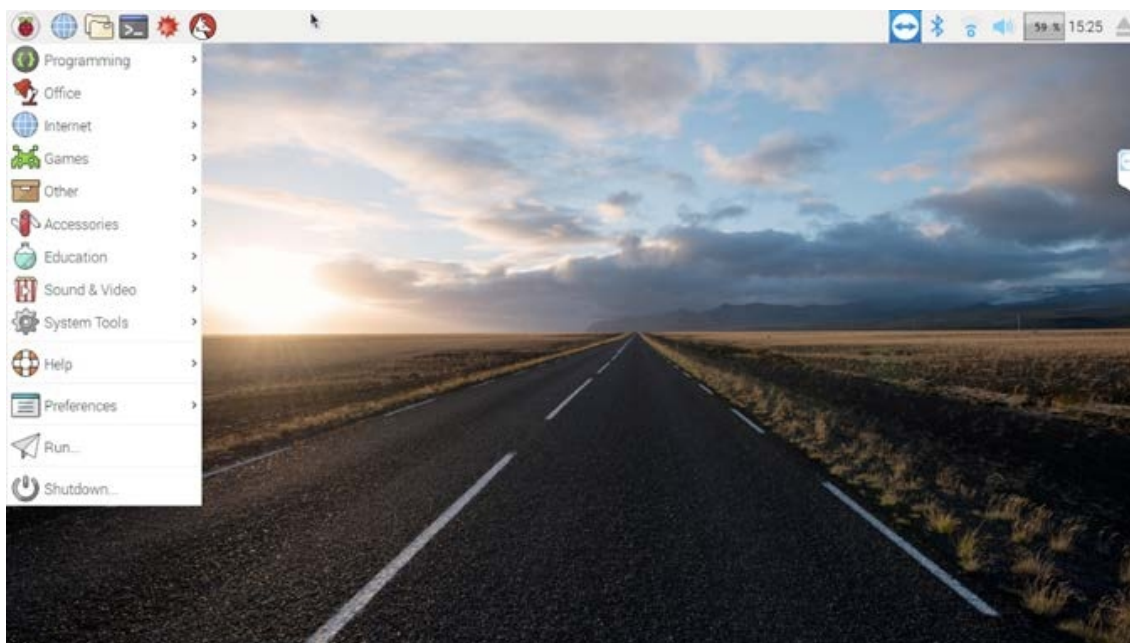


图4-5 Raspbian桌面

1.导航栏左上角

导航栏左上角的**菜单**（Menu）包含了很多应用软件，分为下面几类。

- Programming：编程工具，如动态编程语言Python，用于数学运算的Mathematica，以及用于编程教育的Scratch等。
- Office：办公软件，即开源的LibreOffice套装。
- Internet：互联网软件，如电子邮件客户端和浏览器。
- Games：游戏。这里有点失望，除了Minecraft，就是用于游戏编程的Python Games。
- Accessories：工具软件，如文件管理器File Manager、终端Terminal、文本编辑器等。
- Sound & Videos：VLC播放器。

菜单末端还有两个选项。

- Preferences：系统配置，可以在里面设置时间、语言、显示等选项。
- Shutdown：用于关机或重启。

紧邻着菜单的是来自菜单的五个常用软件，依次是浏览器：用于上网；文件管理器：用于浏览和操作文件；终端：以命令行的方式控制操作系统，本书后面将常用到终端；进行科学运算的Mathematica和Wolfram。

2.导航栏右侧

导航栏右侧通常有树莓派运行状态的几个信息。

- 蓝牙。
- Wi-Fi。
- 声音控制。
- CPU使用监控。
- 时间。

- 可插拔设备，如USB存储器。

你可以点击相应的图标进入设置页面。比如点击Wi-Fi图标，可以选择要连接的无线网络，并输入Wi-Fi密码。

导航栏之外的空间，就是桌面。桌面上有一个**回收箱**（Wastebasket）。此外，你可以把一些文件放在桌面上。如果你打开某个有图形化界面的软件，它的桌面也会显示在桌面上方。接下来，我们来接触两款有图形化界面的软件。

4.4 Scratch

首先，简要地了解Scratch。Scratch是由麻省理工学院开发的一款图形化编程软件。这个软件已经预装在Raspbian中，在“菜单”中找到后点击它，就可以打开，如图4-6所示。

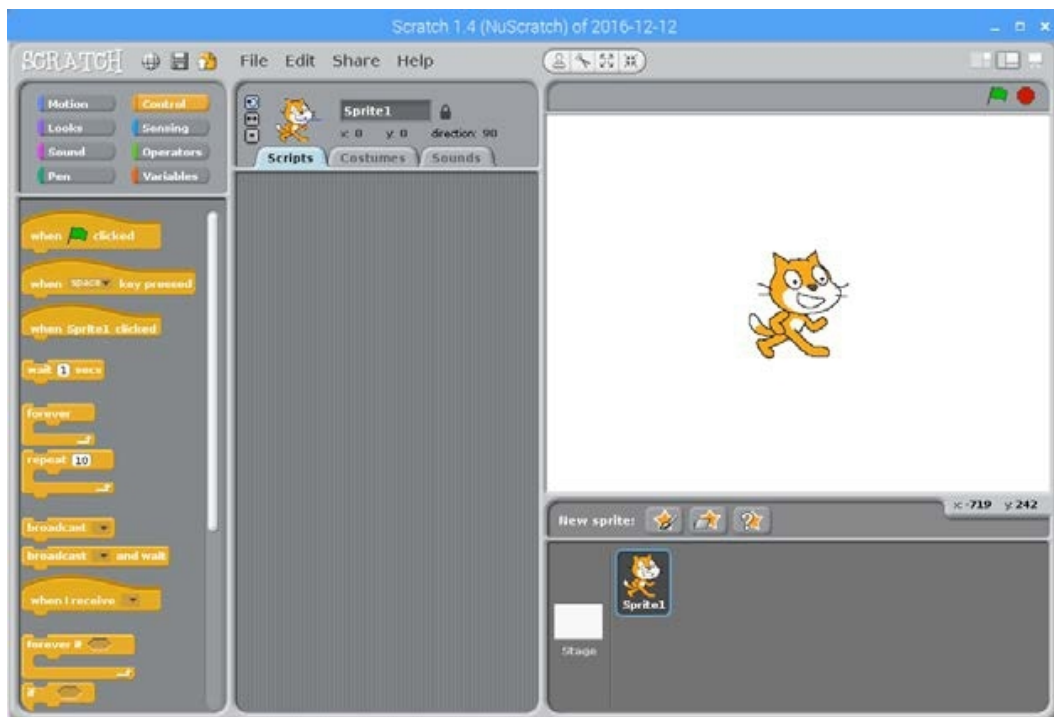


图4-6 Scratch界面

Scratch的界面分为左中右三栏。右边是一只猫，左边是编程可用的命令。把左边的命令拖到中间栏，就可以制作程序。由于左栏提供

了所有可用的命令，所以编程时不需要记住任何语法。因此，Scratch经常用于幼儿教育。

我们把下面几个命令组合在一起，就写成了一个小程序，如图4-7所示。

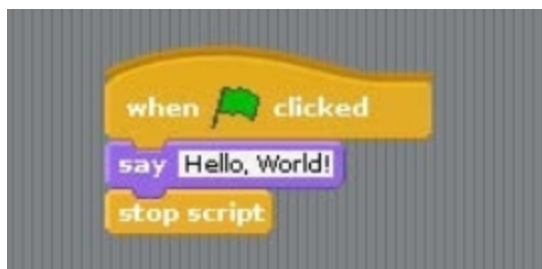


图4-7 Scratch代码

这段程序很直观。当我们点击界面上的旗帜按钮时，猫咪就会说“Hello,World!”。树莓派预装Scratch的目的也很明了：让孩子们尽早享受编程的乐趣。

4.5 K Turtle

Raspbian系统中还可以安装一款叫KTurtle的软件。这款软件有一个图形化界面。整个界面分为左右两栏，如图4-8所示。我们通过左边填写的程序，来控制右栏小海龟的移动。移动的小海龟会在画面上留下行动的轨迹，从而绘制出各种各样的图形。在绘图过程中，小海龟不断移动，左侧程序栏中也会用黄色标明执行到哪一行了，非常有趣。

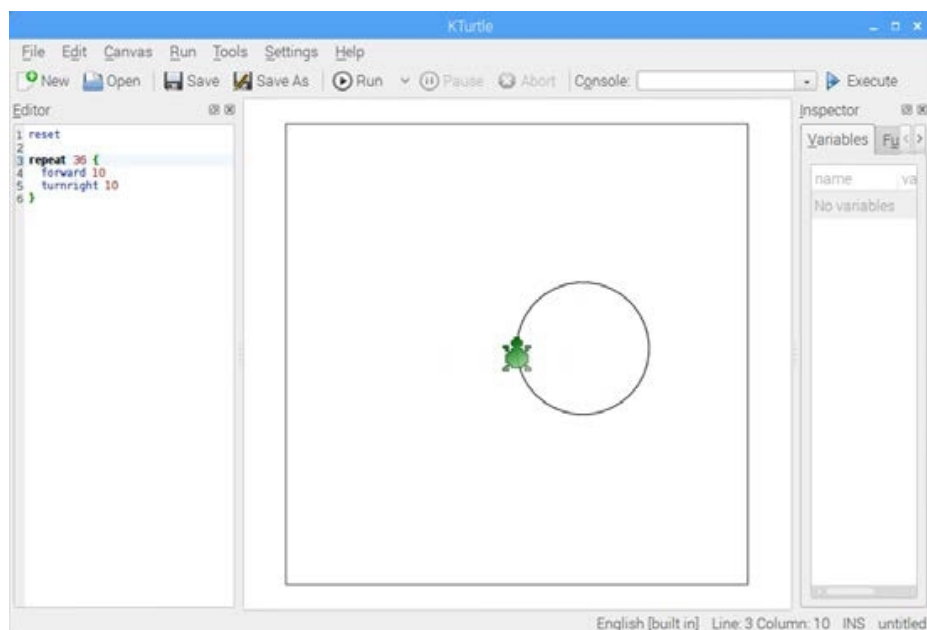


图4-8 K Turtle界面

我们首先来看 K Turtle 的下载安装。从树莓派菜单中打开 Terminal，依次输入以下命令：

```
$sudo apt-get update  
$sudo apt-get install -y kturtle
```

等命令运行完，K Turtle 也就安装好了。

在 Terminal 中输入 kturtle，按 Enter 键打开 K Turtle 界面，就可以开始编程了。常见的控制海龟行动的命令如下所示。

- reset：清空画面。
- pencolor：设置画笔颜色。
- forward：小海龟前进，后面跟一个数字，表示前进的距离。
- backward：小海龟后退，后面跟一个数字，表示后退的距离。
- turnleft：左转，后面跟一个数字，表示左转的角度。
- turnright：右转，后面跟一个数字，表示右转的角度。
- go：让小海龟瞬移到某个位置。后面跟两个数字，表示位置。
- print：在屏幕上打印文字。

- repeat：重复某些动作。后面跟数字和花括号，数字表示重复次数，而花括号中的内容表示重复的动作。

此外，还要一些设置命令。

- reset：清空画面。
- pencolor：设置画笔颜色。
- fontsize：设置字体大小。

程序中的文本和数字等数据可以储存在变量中，以便之后反复使用。变量用\$a的形式表示。KTurtle中更复杂的命令可以通过它的“帮助”菜单来查询。

下面我们用KTurtle绘制一个花朵。在KTurtle的代码框中输入：

```
reset
canvassize 200, 200

learn circle $x {
    penup
    backward $x / 2
    pendown
    repeat 36 {
        forward $x
        turnleft 10
    }
    penup
    forward $x / 2
    pendown
}
```

```

}

learn petal {
    turnright 90
    circle 1
    turnleft 90
}

go 100, 170
direction 0
pencolor 0, 128, 0
penwidth 5
forward 100

direction 90
pencolor 230, 230, 0
for $x = 0.4 to 2.8 step 0.4 {
    circle $x
}

pencolor 190, 0, 0
penwidth 2
penup
forward 4.2
turnright 15
pendown
repeat 6 {
    petal
    turnleft 60
    penup
    forward 16.8
    pendown
}

go 0, 0

```

运行后，小海龟就开始绘制花朵了，如图4-9所示。

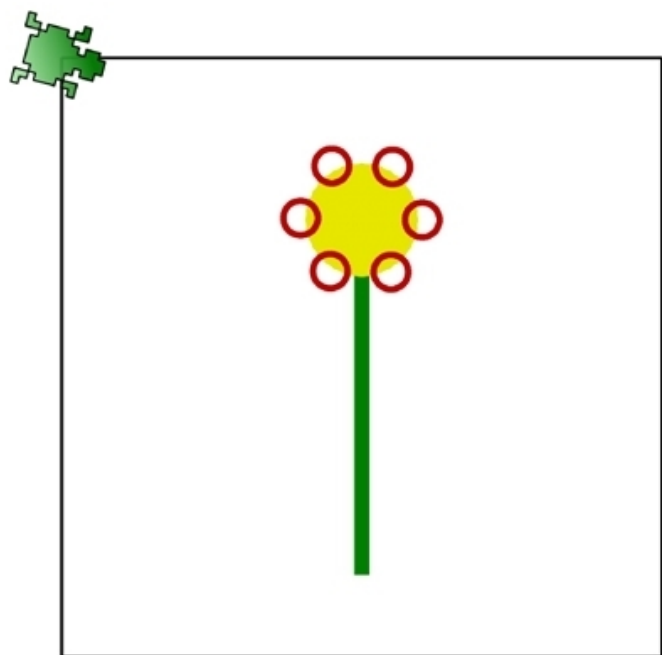


图4-9 绘图结果

第5章 贝壳里的树莓派

树莓派的图形化桌面算不上精美，如果真想用图形化界面进行办公，那么你恐怕会失望。用树莓派的浏览器打开网站上的视频，很可能会遭遇页面加载缓慢和视频播放卡顿等情况。如果想打开多个窗口工作，那么桌面很容易崩溃。毕竟，树莓派的性能不高，而计算机图形的呈现相当消耗资源。幸好，Linux提供了一种更易与树莓派互动的方式——Shell。

5.1 初试Shell

打开终端，桌面上就会出现一个黑色背景的窗口，窗口上显示着：

```
pi@raspberrypi:~ $
```

这里的pi是用户名，raspberrypi是计算机的名字，\$是命令提示符。如果敲击键盘，那么字符会显示在\$提示符的后面，形成一串文本形式的命令。在英文中，Shell是贝壳之类的外壳。在Linux中，所谓的Shell，就是运行在终端中的文本互动程序。Shell分析文本输入，然后把文本转换成相应的计算机动作。用户透过Shell这个“壳”，来触及电脑。贝壳里的Shell，如图5-1所示。

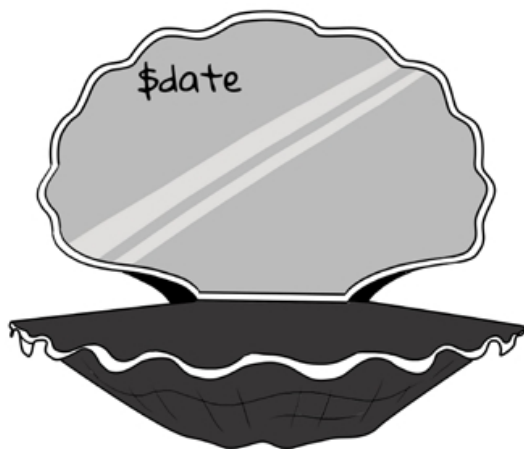


图5-1 贝壳里的Shell

在后面的内容中，将用\$来表示Linux系统Shell的命令提示符，例如输入date命令：

```
$date
```

date用于日期时间的相关功能。按Enter键后，Shell会显示系统的当前时间。

Shell看起来简陋，但实际上比图形化桌面强大得多。Linux操作系统继承自UNIX操作系统。无论是Linux操作系统，还是UNIX操作系统，最初都只提供了Shell这一种用户操作界面。如果你习惯了这种文本操作方式，会渐渐体会到它的好处。

5.2 用命令了解树莓派

为了展示Shell的功能，首先介绍一些命令，用于查询系统信息，例如CPU的型号、内存的大小、IP地址等。这些命令可以让你更加了解树莓派的硬件，另一方面，你也可以借此机会体验一下Shell。

1.Linux通用查询命令

Linux系统提供了各种各样的命令。在Shell中输入这些命令可以实现许多功能。首先用lscpu命令来查询CPU的信息：

```
$lscpu
```

终端窗口中就会打印出CPU的信息：

```
Architecture:      armv7l
Byte Order:         Little Endian
CPU(s):             4
On-line CPU(s) list: 0-3
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s):          1
Model name:         ARMv7 Processor rev 4 (v7l)
CPU max MHz:        1200.0000
CPU min MHz:        600.0000
```

可以看到，这个树莓派用的是4核的ARM处理器，最高频率可以达到1200MHz。

然后，可以用free命令来了解内存的使用状况：

```
$free -h
```

在使用上面的命令时，增加了-h的**选项**（option）。通过给命令增加选项，可以改变命令的行为方式。这里的字母h是human readable的意思。如果不使用-h选项，那么free命令会以字节为单位显示结果。有了-h选项，free可以将结果转换成更适合显示的单位。

Shell打印的结果如下：

	total	used	free	shared	buffers	cached
Mem:	862M	739M	122M	14M	44M	397M
-/+ buffers/cache:		298M	563M			
Swap:	99M	0B	99M			

可以看到，内存总量是862MB，其他列中还显示了已用和可用的内存空间。通过增加选项，Linux命令的功能变得更加丰富。

再看SD卡的存储情况，用命令fdisk：

```
$sudo fdisk -l
```

命令fdisk用于显示磁盘信息。选项-l表示列出所有磁盘。可以看到命令前面增加了sudo。某些命令的运行需要特殊权限，而sudo提供了以系统管理员的身份来执行后面的命令，即fdisk-l。结果的最后两行如下：

Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/mmcblk0p1		8192	131071	122880	60M	c	W95 FAT32 (LBA)
/dev/mmcblk0p2		131072	30318591	30187520	14.4G	83	Linux

整个SD卡被分成了两个分区，其中的一个分区有60MB，专门用于树莓派的开机启动；另一个分区用于储存其他的所有数据。

使用lsusb，可以找到所有的USB外设：


```
$lsusb
```

Shell将打印：

```
Bus 001 Device 005: ID 0e8f:2517 GreenAsia Inc.  
Bus 001 Device 006: ID 045e:0750 Microsoft Corp. Wired Keyboard 600  
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast  
Ethernet Adapter  
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.  
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

使用uname命令，可以打印出操作系统的信息：

```
$uname -a
```

选项-a表示显示所有的相关信息。Shell将打印：

```
Linux raspberrypi 4.1.19-v7+ #858 SMP Tue Mar 15 15:56:00 GMT 2016 armv7l  
GNU/Linux
```

这里的系统使用的内核是Linux 4.1.19版本，而内核的发布时间是2016年3月15日。

最后，用ifconfig命令来查看网络接口：

```
$ifconfig
```

命令运行结果如下：

```
eth0    Link encap:Ethernet HWaddr b8:27:eb:d8:ed:f4
        inet6 addr: fe80::9b8b:c0de:d083:6ddd/64 Scope:Link
        UP BROADCAST MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING MTU:65536 Metric:1
        RX packets:243 errors:0 dropped:0 overruns:0 frame:0
        TX packets:243 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:19020 (18.5 KiB) TX bytes:19020 (18.5 KiB)

wlan0   Link encap:Ethernet HWaddr b8:27:eb:8d:b8:a1
        inet addr:192.168.0.108 Bcast:192.168.0.255 Mask:255.255.255.0
        inet6 addr: fe80::ba27:ebff:fe8d:b8a1/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:45926 errors:0 dropped:4268 overruns:0 frame:0
        TX packets:10469 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:26867855 (25.6 MiB) TX bytes:1360267 (1.2 MiB)
```

其中eth0代表了以太网接口，wlan0代表了Wi-Fi接口，而lo是虚拟出来的本地接口，用来表示本机。在连接上网的接口中，我们可以看到该接口的IP地址等信息。例如wlan0的IP地址是192.168.0.108。因为没有插网线，所以eth0并没有IP地址。

2.树莓派专用查询命令

除通用的Linux命令外，树莓派还提供了vcgencmd命令，用于和树莓派硬件直接互动。比如在Shell中执行：

```
$vcgencmd measure_temp
```

在上面的命令中，第二段的measure_temp是命令的参数。参数是选项之外另一种给命令提供额外信息的方式。上面的命令将返回CPU的温

度：

```
temp=51.5'C
```

用下面的命令测量树莓派的核心电压：

```
$vcgencmd measure_volts core
```

返回电压值：

```
volt=1.2000V
```

5.3 什么是Shell

Shell是UNIX系统提供的文本交互界面。你只要用键盘输入命令就可以和操作系统交互，但这还是不够具体。说到底，Shell其实是一个运行着的程序。这个程序接收到你两次单击“Enter”键之间的输入，就会对输入的文本进行分析，比如下面这个命令：

```
$free -h
```

包括空格在内总共7个字符。Shell程序通过空格区分出命令的不同部分。第一个部分是命令名，剩下的部分是选项和参数。在这个例子中，Shell会进一步分析第二个部分，发现这一部分的开头是“-”字符，从而知道它是一个选项。

有了命令名，Shell下一步就要执行该命令名对应的动作。这听起来就像是在戏剧舞台上，演员按照脚本演戏。Shell命令可以分为如下三类。

- Shell 内建函数（built-in function）。
- 可执行文件（executable file）。
- 别名（alias）。

Shell的内建函数是保存在Shell内部的脚本。相对应地，可执行文件是保存在Shell之外的脚本。Shell必须在系统中找到对应命令名的可执行

文件，才能正确执行。我们可以用绝对路径来告诉Shell可执行文件所在的位置。所谓路径，是指一个文件在存储空间的位置，例如：

```
/bin/date
```

这个路径表明date这个可执行文件位于根目录下的bin文件夹内。

如果用户只给出了命令名，而没有给出准确的位置，那么Shell必须自行搜索一些特殊的位置，也就是所谓的默认路径。Shell会执行第一个和命令名相同名字的可执行文件。这就相当于，Shell帮我们自动补齐了可执行文件的位置信息。我们可以通过which命令来确定命令名对应的是哪个可执行文件：

```
$which date
```

别名就是给某个命令起的一个简称，以后在Shell中就可以通过这个简称来调用对应的命令。在Shell中，我们可以用alias来定义别名：

```
$alias freak="free -h"
```

Shell会记住我们的别名定义。以后在这个Shell中输入命令freak时，都将等价于输入free-h。

在Shell中，可以通过type命令来了解命令的类型。如果一个命令是可执行文件，那么type将打印出文件的路径。

```
$type date
$type pwd
```

总的来说，Shell就是根据空格和其他特殊符号，来让电脑理解并执行用户要求的动作。到了后面，我们还将看到Shell中的其他特殊符号。

5.4 Shell的选择

Shell是文本解释器程序的统称，所以包括了不止一种Shell。常见的Shell有sh、bash、ksh、rsh、csh等。在树莓派中，就安装了sh和bash两个

Shell解释器。sh的全名是Bourne Shell，其中Bourne就是这个Shell的作者。而bash的全名是Bourne Again Shell。在UNIX系统中流行的是sh，而bash作为sh的改进版本，提供了更加丰富的功能。一般来说，都推荐使用bash作为默认的Shell。树莓派及其他Linux系统中广泛安装了sh，都是出于兼容历史程序的目的。

我们可以通过下面的命令来查看当前的Shell类型：

```
$echo $SHELL
```

echo用于在终端打印文本，而\$是一个新的Shell特殊符号，它提示Shell，后面跟随的不是一般的文本，而是用于存储数据的变量。Shell会根据变量名找到真正的文本，并替换到变量所在的位置。SHELL变量存储了当前使用的Shell的信息，可以在bash中用sh命令启动，用exit命令从中退出。

5.5 命令的选项和参数

我们已经看到，一行命令里还可以包含着选项和参数。总的来说，选项用于控制命令的行为，而参数说明了命令的作用对象，例如：

```
$uname -m
```

在上面的命令中，选项-m影响了命令uname的行为，导致uname输出了树莓派的CPU型号。如果不是受该选项的影响，那么uname输出的将是Linux。我们不妨把每个命令看作多功能的瑞士军刀，而选项使命令可以在不同的功能间切换。由一个“-”引领一个英文字母，这称为短选项。多个短选项的字母可以合在一起，跟在同一个“-”后面。比如，下面的两个命令是等价的：

```
$uname -m -r  
$uname -mr
```

此外还有一种长选项，是用“--”引领一个英文单词，比如：

```
$date --version
```

上面的命令将输出date程序的版本信息。

如果说选项控制了瑞士军刀的行为，那么参数就提供了瑞士军刀发挥用途的原材料。以echo命令为例，它能把字符打印到终端。它选择打印的对象，正是它的参数：

```
$echo hello
```

有的时候，选项也会携带变量，以便说明选项行为的原材料，比如：

```
$sudo date --set="1999-01-01 08:00:00"
```

选项“--set”用于设置时间，用等号连接的就是它的参数。date会把日期设置成这一变量所代表的日期。如果用短选项，那么就要用空格取代等号了：

```
$sudo date -s "1999-01-01 08:00:00"
```

值得注意的是，Shell对空格敏感。当参数信息中包含了空格时，我们需要用引号把参数包裹起来，以便Shell能识别出这是一个整体。

选项和参数都是提供给命令的附加信息，因此，命令最终会拿这些字符串做什么，是由命令自己决定的。因此，有时会发现一些特异的选项或参数用法。这个时候，就要从文档中寻找答案。

5.6 如何了解一个陌生的命令

每一个Linux系统都带有一套完善的文档用于解释每个命令的用途。你可以用下面三个命令来调用某个命令的文档信息。

1.whatis

```
$whatis ls
```

whatis命令的作用是用很简短的一句话来介绍命令。

2.man

```
$man ls
```

man会返回命令的帮助手册。对于大部分Linux自带的命令来说，作者编写时，都会写一个帮助文档，告诉用户怎样使用这个命令。man可以说是我们了解Linux最好的百科全书，它不仅告诉你Linux自带的命令的功能，还可以查询Linux的系统文件和系统调用。如果想要深入学习Linux，就必须懂得如何用man来查询相关文档。

3.info

```
$info ls
```

info将返回更详细的帮助信息。

5.7 Shell小窍门

使用Shell时应用一些小窍门，可以让你事半功倍。

1.命令补齐

大多数的Shell都有命令补齐的功能。当你在\$的后面输入命令的一部分时，比如“dat”，按Tab键，Linux会把它补充成为“date”。在这个过程中，Shell会搜索该命令名的所有可能。如果只有一种可能，那么Shell就会把该文件名补齐。如果不止一种，那么第一次按Tab键会没有反应，第二次按Tab键时，终端会打印出所有可能的命令名。比如输入“da”，按两次Tab键后，终端输出：

```
dash date
```

这样的提示，能帮你想起自己想要输入的命令。

2.文件名补齐

不止是命令名，如果输入的是作为参数的文件名，Linux也可以帮你补齐。比如，当前目录下有a.txt文件。当你输入到ls a.t的时候，按Tab

键，Shell会帮你补齐该文件名，即`ls a.txt`。

3.历史命令

在Shell中，可以用向上箭头，或`history`命令来查看之前输入的命令。

```
$history
```

4.中止与暂停命令

当一个命令运行时，如果中途想要停止它，那么可以用快捷键Ctrl+C。如果只是想暂时停止，那么可以使用快捷键Ctrl+Z。中止与暂停应用了Linux中的**信号**（Signal）机制，笔者将在第23章介绍。

第6章 好编辑

人类经常用文字来表达和交流抽象的信息。人们使用键盘向电脑输入的大多是文字。计算机学科通常把一串文字构成的信息称为**文本**（Text）。很多用户使用计算机的主要目的就是编辑文本：写公文、写电子邮件、写小说……精通计算机的程序员，也往往以文本形式的程序向计算机表达自己的意志。在Linux系统下，很多配置文件也是人类可读的文本。在计算机的用户软件中，一定有文本编辑器的一席之地，文本编辑器可以创建、修改和保存文本。

6.1 图形化的文本编辑器

Raspbian中包含了一个有图形化界面的文本编辑器，也就是菜单中的Text Editor，如图6-1所示。这个文本编辑器有一个像白纸一样的界面。敲击键盘，那些敲击的字符就会“写”在这张白纸上。用键盘的上下左右键，或者用鼠标移动光标，就可以改变光标的位置。在光标所在的位置，可以插入或删除文本。如果你用过“记事本”或Word这样的软件，那么对这类编辑文本的过程肯定早有经验。

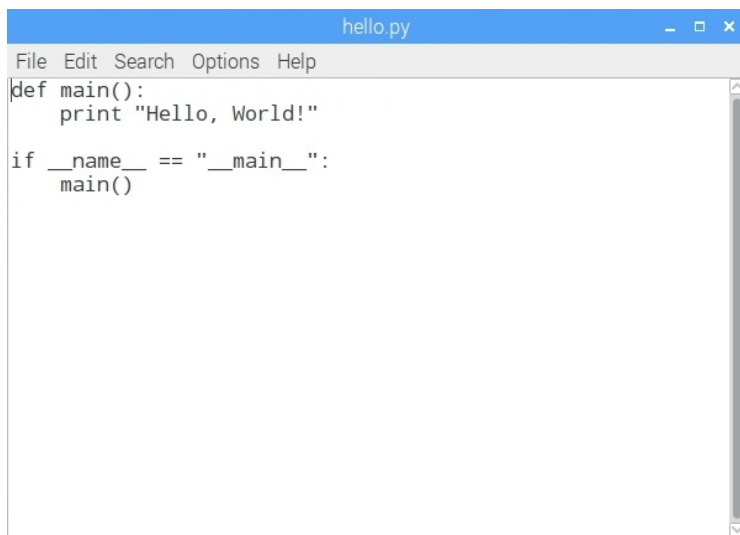


图6-1 图形化的文本编辑器

常见的操作如复制粘贴，在文本编辑器的Edit（编辑）菜单里都可以找到，从上到下一共有7个按钮，如下所示。

- Undo：撤销上一个操作，快捷键Ctrl+Z。
- Redo：重做上一个撤销的操作，快捷键Shift+Ctrl+Z。
- Cut：剪切选中的内容，快捷键Ctrl+X。
- Copy：复制选中的内容，快捷键Ctrl+C。
- Paste：粘贴剪贴板的内容到光标处或替换选中的内容，快捷键Ctrl+V。
- Delete：删除选中的内容，功能同键盘上的Delete键。
- Select All：全选，快捷键Ctrl+A。

文本编辑完成后就可以保存文本了。Linux以文件的形式存储文本。文本最终以文件为单位存储在Micro SD卡上。选择File（文件）菜单中的Save选项来保存文件，在弹出的窗口中输入文件名，如 **hello.txt**，选择你希望保存的路径，单击Save按钮后保存文件，如图6-2所示。

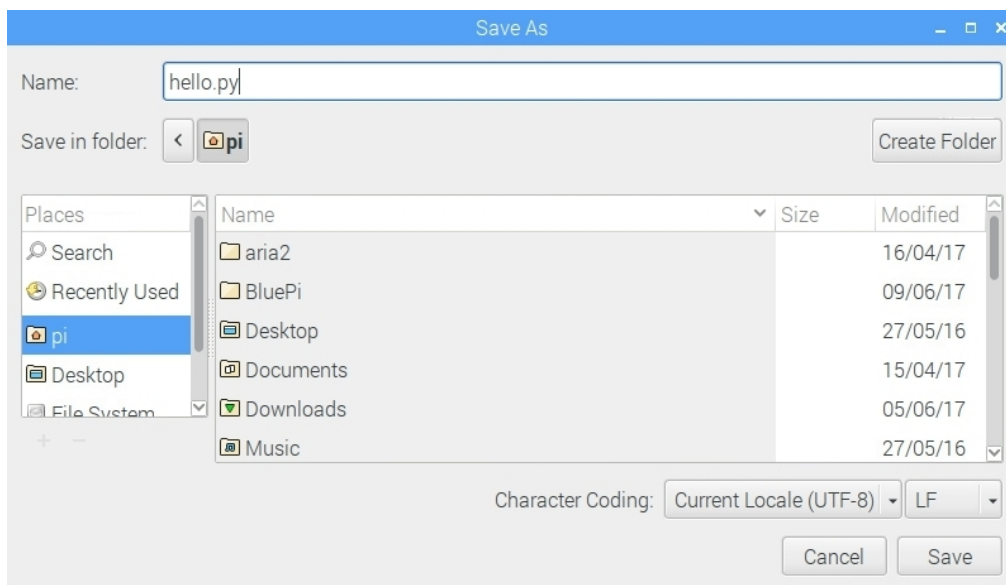


图6-2 Save/Save As（保存/另存为）对话框

在树莓派的**文件管理器**（File Manager）中找到该文件，在右键快捷菜单中选择Text Editor选项，即可用文本编辑器打开它，如图6-3所示。

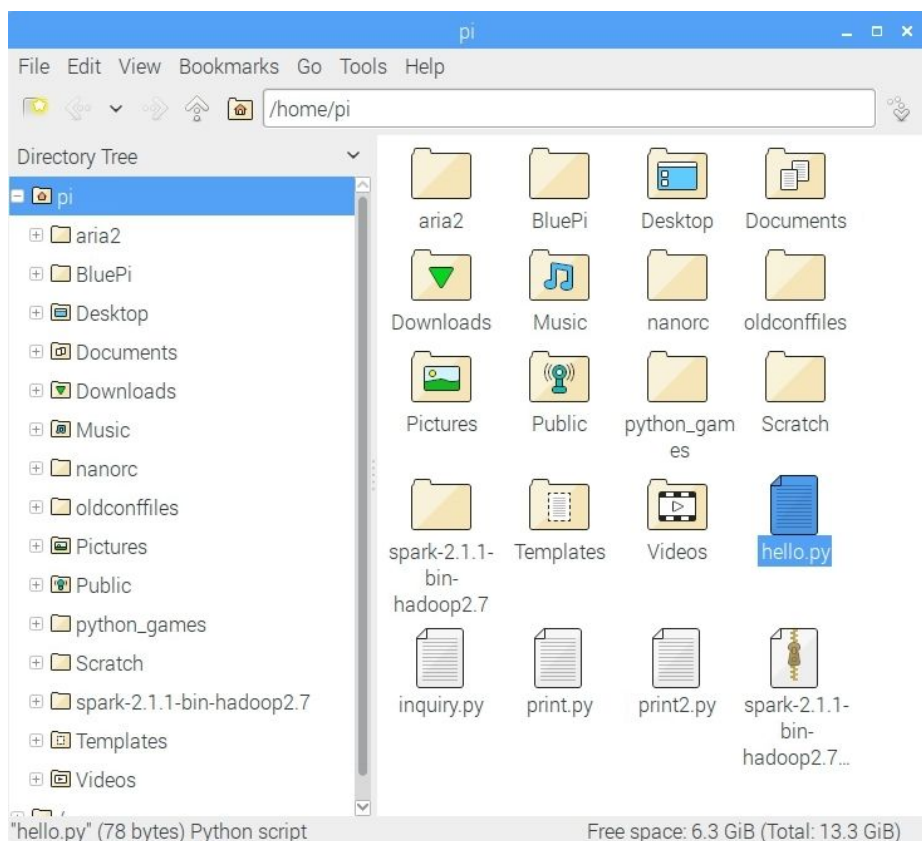


图6-3 在文件管理器中找到文件

6.2 使用nano

GNU nano是Shell中常用的一款文本编辑器，它以简单易用著称。其实，在Shell下，还有更出名、功能更强大的Vi和Emacs编辑器，但这两款编辑器的学习曲线都比nano陡峭很多。因为nano对于一般的文本编辑来说已经够用，所以这里着重介绍nano编辑器。

在Shell中输入下面的命令就可以启动nano：

```
nano test.txt
```

命令nano后面跟着想要修改的文件名。如果当前文件夹下存在名为 **test.txt** 的文件，则该命令将打开这个文件。否则，命令nano会创建一个新文件。随后，Shell会进入nano的编辑界面。nano的编辑方式和图形化的记事本工具类似，也是“所见即所得”。用上下左右键就可以把光标移动到想要编辑的位置，然后输入或删除即可。

完成之后，可以按快捷键Ctrl+O来保存文件。nano会询问你是否保存缓存中的修改：

```
Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?
```

输入Y并将改动存入文件，此时nano会让你再次确认存入文件的文件名：

```
File Name to Write: test.txt
```

按Enter键确认后，修改将存入 *test.txt* 文件。随后，按快捷键Ctrl+X可以退出nano，重新回到Shell的命令行。

nano中的很多操作都是通过功能键实现的。上面保存文件用的快捷键Ctrl+O，就是一个功能键。因为功能键很多，所以背起来很痛苦。万幸的是，nano界面（如图6-4所示）的最下方会给出功能键的提示。

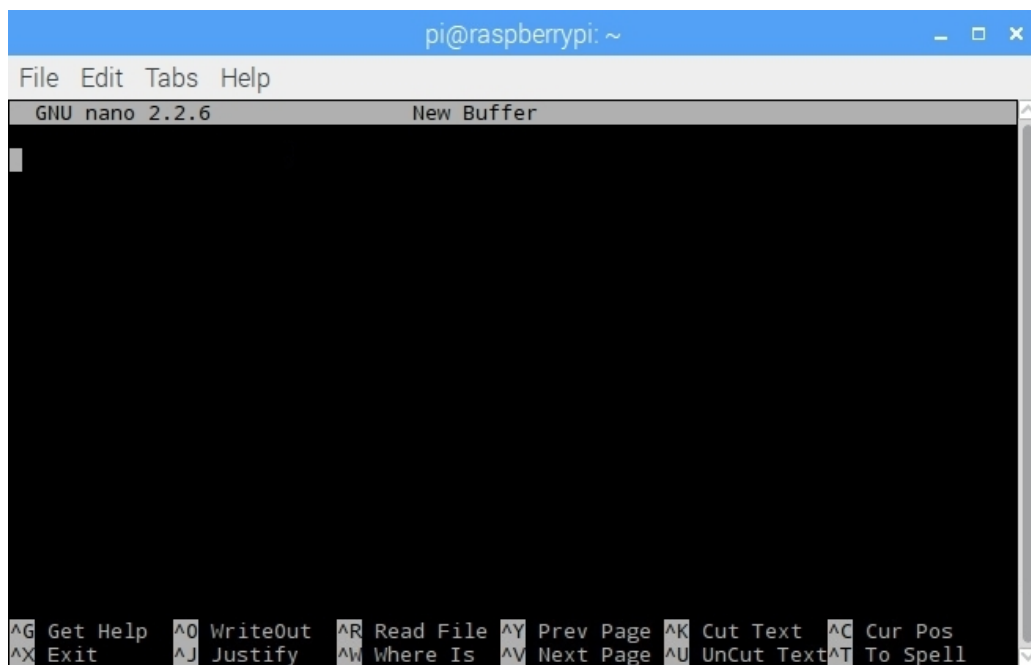


图6-4 nano界面

在提示中，^表示Ctrl键，M表示Alt键。因此，^G表示的就是同时按下Ctrl键和G键。下面是一些常用的功能键。

- M-\，把光标移到文本开始。
- M-/，把光标移到文本结尾。

- M-A, 开始选择文本块。
- ^K, 剪切所在行或选定的文本块。
- M-6, 复制所在行或选定的文本块。
- ^U, 粘贴。
- ^G, 帮助。

6.3 语法高亮

nano可以支持语法高亮, 从而更好地服务于编程。为了使语法高亮, 首先要安装语法高亮文件:

```
$git clone https://github.com/nanorc/nanorc.git
$cd nanorc/
$make install
```

安装完成后, 可以看到 `~/.nano/syntax` 下多了很多语法高亮文件:

ALL.nanorc	go.nanorc	markdown.nanorc	ruby.nanorc
awk.nanorc	html.nanorc	mpdconf.nanorc	sed.nanorc
c.nanorc	ini.nanorc	nanorc.nanorc	shell.nanorc
cmake.nanorc	inputrc.nanorc	nginx.nanorc	sql.nanorc
coffeescript.nanorc	java.nanorc	patch.nanorc	systemd.nanorc
colortest.nanorc	javascript.nanorc	peg.nanorc	tex.nanorc
csharp.nanorc	json.nanorc	php.nanorc	vala.nanorc
css.nanorc	keymap.nanorc	pkg-config.nanorc	vi.nanorc
cython.nanorc	kickstart.nanorc	pkgbuild.nanorc	xml.nanorc
default.nanorc	ledger.nanorc	po.nanorc	xresources.nanorc
dot.nanorc	lisp.nanorc	privoxy.nanorc	yaml.nanorc
email.nanorc	lua.nanorc	properties.nanorc	yum.nanorc
git.nanorc	makefile.nanorc	python.nanorc	
glsl.nanorc	man.nanorc	rpmspec.nanorc	

每个文件代表了对一种语言的语法高亮支持。比如 `python.nanorc`, 就包含了对Python语言的语法高亮支持。将语法高亮文件添加到 `~/.nanorc` 中, 就能让nano启动对相应语言的语法高亮支持, 例如:

```
include ~/.nano/syntax/c.nanorc
include ~/.nano/syntax/css.nanorc
include ~/.nano/syntax/java.nanorc
include ~/.nano/syntax/makefile.nanorc
include ~/.nano/syntax/php.nanorc
include ~/.nano/syntax/python.nanorc
include ~/.nano/syntax/ruby.nanorc
include ~/.nano/syntax/tex.nanorc
include ~/.nano/syntax/xml.nanorc
```

只要有需要，把相应的语法高亮文件加入 *.nanorc* 中，就能实现该语言的语法高亮。这时再打开获得支持的程序文本，就可以看到语法高亮的效果。图6-5是用nano打开了一段Python程序。

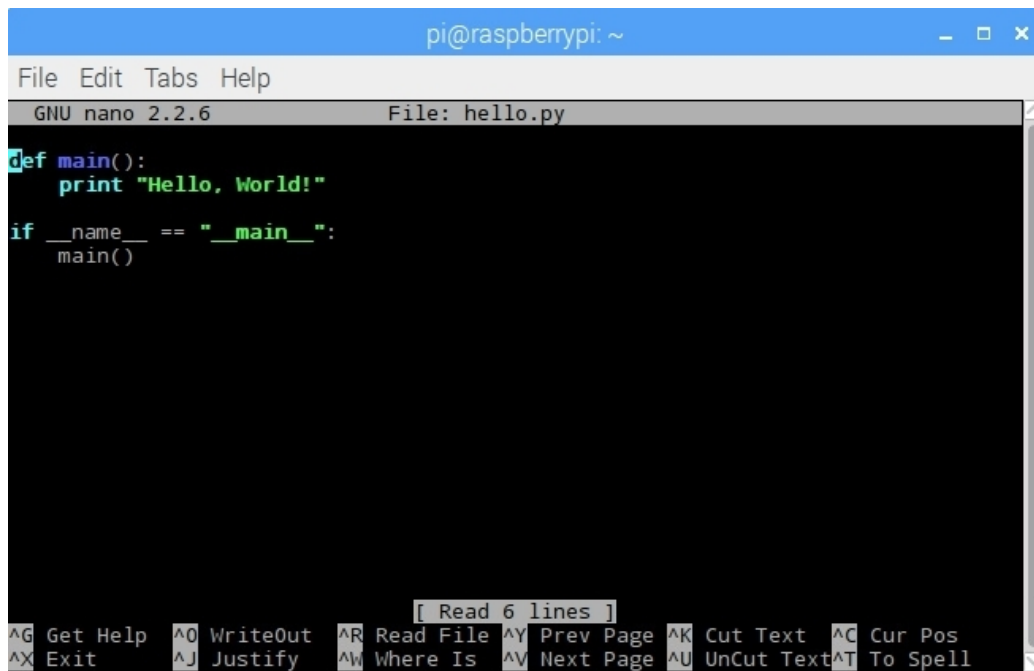


图6-5 nano中Python程序的语法高亮

在nano中，使用M-Y功能键可以开关语法高亮功能。

6.4 文件基础操作

用nano编辑文件并保存后，当前**目录**（Directory）下就会出现一个新的文件，文件名就是我们使用时的文件名。所谓的目录，就是一个类

似于“文件夹”的收纳盒，其中可以包含多个文件。用下面的命令可以显示Shell当前目录下的文件：

```
$ls
```

文件是Linux进行数据存储的唯一形式。除了用户编辑生成的文本，数据还可能是Linux系统中的程序或配置文件。就连硬件设备，也会虚拟成一个文件。既然文件的地位如此重要，Linux中自然少不了用于文件操作的命令，比如复制文件：

```
$cp test.txt test_again.txt
```

复制之后，同一目录下会出现一个名为 *test_again* 的文件。这个文件中包含的文本和 *test.txt* 相同。

又如删除文件的rm：

```
$rm test.txt
```

删除文件之后，该目录下的 *test.txt* 就消失了。

还有移动文件：

```
$mv test_again.txt test.txt
```

当前目录下的 *test_again.txt* 文件移为 *test.txt* 文件。这里的移动，等价于重命名的功能。

用nano保存文件后，如果没有说明目录，那么文件就保存在当前目录下。我们可以用下面的命令来查询Shell所在的当前目录：

```
$pwd
```

该命令输入后，Shell显示的是：

```
/home/pi
```

这串文字说明了当前目录在当前文件系统中的位置，即 */home/pi* 目录。

一个目录下的文件不能重名。因此，在 ***/home/pi*** 这样的目录下加上文件名，就能唯一确定这个文件。这就是文件的**路径**（path）。比如：

```
/home/pi/test.txt
```

在命令中，我们也可以用这个路径来唯一指代要操作的对象，比如用nano打开文件：

```
$nano /home/pi/test.txt
```

或者删除文件：

```
$rm /home/pi/test.txt
```


第7章 更好的树莓派

拿到树莓派后，需要进行一些初始化配置，以便使用起来更加方便。除此之外，可能需要安装一些软件，以便树莓派能实现更加强大的功能。

7.1 常见初始化配置

树莓派系统的一般用户配置可以通过图形化的配置窗口完成。在菜单中选择Preferences选项，就可以看到配置窗口。配置窗口的界面如图7-1所示。

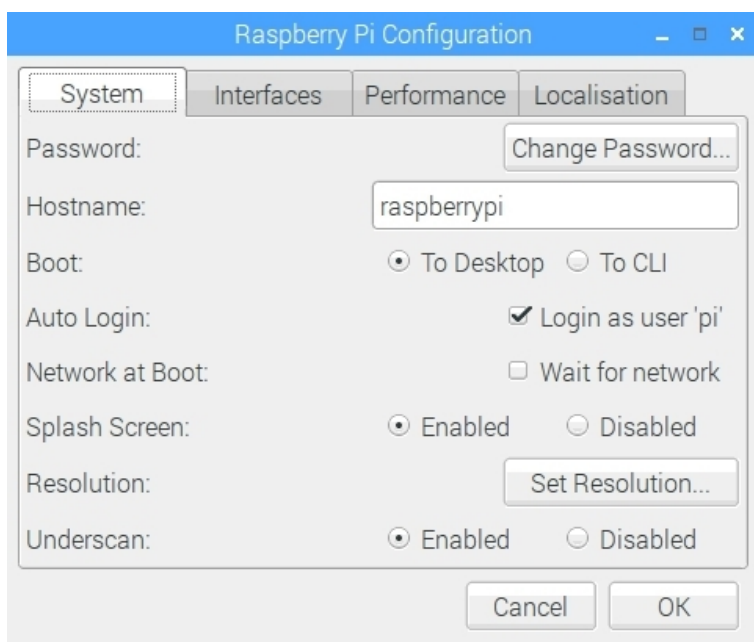


图7-1 Raspberry Pi配置窗口

命令行的配置工具提供了更加丰富的配置功能。在Shell中通过下面的命令可以进入命令行配置工具：

```
$sudo raspi-config
```

Shell中弹出的配置页面如图7-2所示。

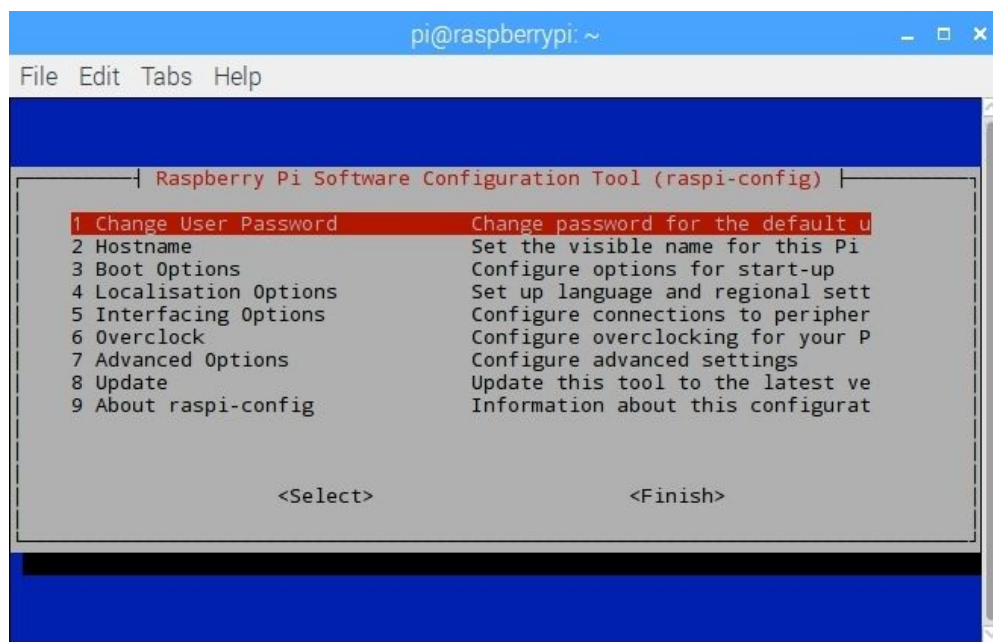


图7-2 Raspberry Pi命令行配置

此外，Linux中还可以通过命令或修改配置文件来改变相关配置，下面列举一些树莓派上的常见配置。

1.配置密码

树莓派的默认用户名是pi，没有密码。这意味着别人可以随意使用你的树莓派。你可以在终端中为pi用户配置密码：

```
$sudo passwd pi
```

2.配置Locale

当打开终端时，终端有可能提醒你Locale未配置。在命令行配置页面中，在“5 Internationalisation Options”→“11 Change Locale”页面下选择Locale选项即可。如果不用图形化界面，那么也可以通过修改 */etc/default/locale* 进行手动配置，在该文件末尾附加：

```
LANG=en_GB.UTF-8
LC_ALL=en_GB.UTF-8
LANGUAGE=en_GB.UTF-8
```

3. 键盘布局

给树莓派连上键盘后，你可能发现键盘和输入字符对应不上，这个时候需要把键盘布局变为美式布局。在命令行配置页面中，在“5 Internationalisation Options”→“I3 Change Keyboard Layout”页面下进行选择即可。

键盘布局也可以通过编辑配置文件进行手动修改。在文件 */etc/default/keyboard* 中找到XKBLAYOUT打头的一行，修改为：

```
XKBLAYOUT="us"
```

4. Wi-Fi连接

你可以点击桌面右上角的Wi-Fi图标，在其中配置Wi-Fi连接，也可以通过修改配置文件来配置Wi-Fi连接。打开配置文件：

```
$sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
```

在其中加入Wi-Fi的ssid和密码：

```
network={
    ssid="Vamei"
    psk="vamei"
}

network={
    ssid="raspberry-pi"
    psk="pipi12345"
}
```

5. 更新固件

树莓派上有许多硬件，如Wi-Fi适配器、蓝牙适配器等。这些硬件都有特定的固件支持。有时候树莓派安装的是比较旧的固件，可能会带来一些问题。因此，你可以从命令行更新固件：

```
$sudo rpi-update
```

7.2 软件升级与安装

我们说托瓦兹是“Linux之父”，是因为他编写并维护着Linux最核心的程序，即Linux内核。除了内核，Linux还需要很多应用程序，比如sh和bash。Linux内核加上应用程序，就构成了一个Linux发行版本。因此，就有不同发行版本的Linux，如Debian、Red Hat、Ubuntu、Raspbian。此外，除了预装的应用程序，用户还需在使用过程中增加新的应用程序。用户可以直接在网上下载程序的源代码，然后自行编译成软件。但编译软件需要很多配置，不同软件之间又有依赖关系，所以普通用户很容易犯错。

为了解决这个问题，Linux发行版本都有软件分发机制。你可以从互联网的软件服务器上找到自己需要的软件并下载安装。这些软件服务器被称为软件源。软件源提供的软件是已经编译好的。如果这些软件依赖于其他的软件，分发系统也会帮助你自动下载。Raspbian继承自Debian，沿用了Debian的软件分发机制。在大部分情况下，你可以通过apt-get命令下载已经编译好的软件。

首先，树莓派需要知道软件源中提供了哪些软件。用下面的命令可以更新软件源，获得最新的软件列表：

```
$sudo apt-get update
```

升级已安装的软件：

```
$sudo apt-get upgrade
```

安装软件，比如MySQL：

```
$sudo apt-get install mysql
```

如果不再需要某个软件，或者某个软件出现了问题，那么可以删除该软件：

```
$sudo apt-get remove mysql
```

上面的apt-get remove不会删除配置文件。为了更彻底地删除软件，可以使用：

```
$sudo apt-get purge mysql
```

修改软件源服务器。有时树莓派官方的软件源下载起来特别慢，这时可以尝试使用国内的镜像。这些镜像服务器或许能提供更快的服务。修改 ***/etc/apt/sources.list*** 内容为：

```
deb http://mirrors.ustc.edu.cn/raspbian/raspbian jessie main contrib non-free  
rpi  
deb-src http://mirrors.ustc.edu.cn/raspbian/raspbian/ jessie main contrib non-  
free rpi
```

这里把软件源服务器修改成中科大的镜像服务器。

第8章 漂洋过海连接你

我们之前使用树莓派的方法，就是给它连上显示器、键盘和鼠标，然后像使用一台普通电脑一样使用它。但很多时候，我们把体积小巧的树莓派当作一台便携设备来使用。这时用户可不希望随身带着体积庞大的鼠标、键盘和显示器。如果能用手中的电脑直接连接树莓派，然后用该电脑的输入输出设备来操纵树莓派电脑，就可以省去很多不必要的麻烦。除此之外，树莓派在物联网情境下的应用，也离不开多样的远程连接方式。本章介绍连接树莓派的其他方式。

8.1 局域网SSH登录

常见的家庭或办公网络都是以一个Wi-Fi路由器为中心的。在这种局域网场景下，可以很容易地用SSH的方式远程登录树莓派。SSH是用于远程服务器管理的加密协议。SSH分为服务器和客户端两部分。树莓派将作为服务器端，而同一局域网中的另一台电脑可以作为客户端。客户端成功登录之后，我们可以从客户端用命令行的方式来远程操作服务器端。

首先，我们需要开启树莓派上的SSH服务器。树莓派已经预装好了SSH服务器，我们只需进入树莓派的设置页面开启即可。从终端用命令行进入设置页面：

```
$sudo raspi-config
```

然后在“5 Interfacing Options”→“P2 SSH”页面中打开SSH服务器。

为了远程连接，我们必须知道树莓派的IP地址。在树莓派上，可以用ifconfig命令来找到树莓派的IP地址：

```
$ifconfig
```

从ifconfig的输出中找到树莓派在局域网中的IP地址。比如ifconfig输出中给出了对应Wi-Fi连接的wlan0端口地址为192.168.1.101。这时就可以用同一局域网下的其他电脑来登录树莓派了。如果这台电脑是Mac OS X或Linux系统的，则可以直接使用ssh命令：

```
$ssh pi@192.168.1.101
```

输入用户pi的密码，就可以远程登录树莓派了。其实使用SSH客户端时，除了说明树莓派的IP地址，还需要一个端口号。在省略端口号时，客户端默认为端口22。

在Windows下，可以使用PuTTY这样的SSH客户端软件。先访问PuTTY官网，下载PuTTY客户端。解压后，单击文件夹中的PUTTY.EXE文件将会打开配置窗口，如图8-1所示。输入树莓派的相关信息来远程登录，PuTTY会弹出一个Shell界面，如图8-2所示。你可以通过这个Shell界面远程操纵树莓派。

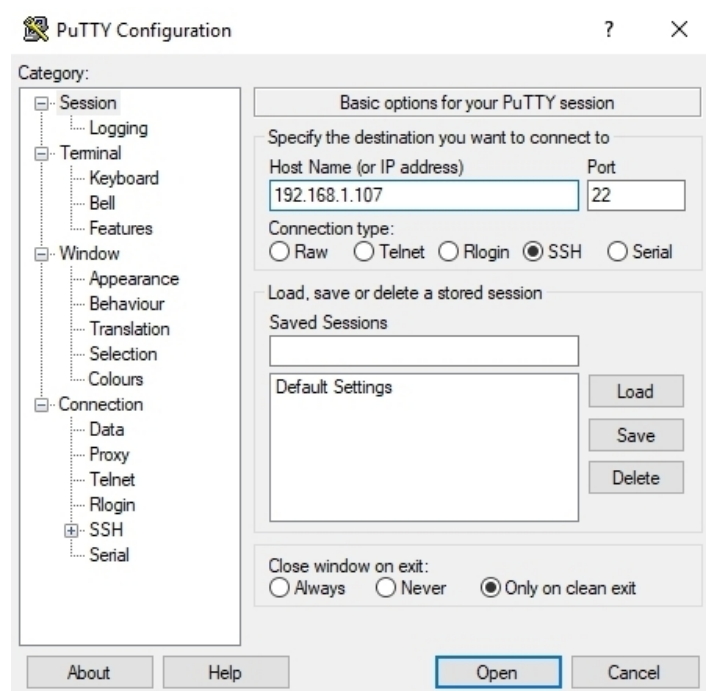
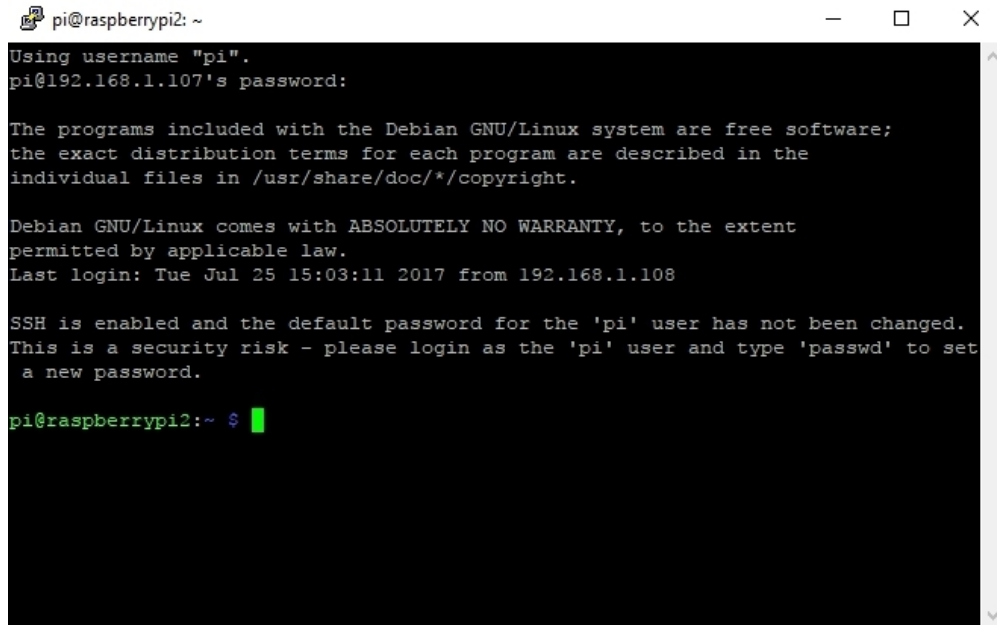


图8-1 PuTTY的配置窗口



```
pi@raspberrypi2: ~
Using username "pi".
pi@192.168.1.107's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jul 25 15:03:11 2017 from 192.168.1.108

SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set
a new password.

pi@raspberrypi2:~ $
```

图8-2 PuTTY登录后的Shell

8.2 Bonjour

在上面的过程中，我们必须从树莓派本地运行ifconfig来查找它的IP地址，给远程登录增加了不必要的麻烦。我们可以用局域网扫描工具来找到树莓派的IP地址。UNIX系统下提供了arp命令行工具，通过ARP协议来找到局域网下所有设备的MAC地址和对应的IP地址。此外，在不同的平台下也有很多图形化的局域网扫描软件，例如iPhone上的Fing、Mac OS X下的LanScan、跨平台的Angry IP Scanner，都可以帮助你列出同一局域网下所有设备的MAC地址和对应的IP。此外，你还可以登录路由器的管理页面。很多路由器都会列出连接设备及其IP。当然，通过这种方式得到的IP是一个列表，还要从中筛选出目标IP。如果局域网中的设备较多，其过程会比较烦琐。

更方便的，树莓派提供了对Bonjour的支持。Bonjour用于自动发现网络上的设备，可以实现局域网上的自动域名解析。在同一局域网下，可以用主机名.local的形式，找到对应的IP地址。由于树莓派的默认主机名是raspberrypi，因此可以用raspberrypi.local来登录树莓派：

```
ssh pi@raspberrypi.local
```


如果局域网内有多台以raspberrypi为名的主机，那么Bonjour将依次把它们称呼为：

```
raspberrypi  
raspberrypi-2  
raspberrypi-3  
.....
```

为了避免主机名的冲突，还可以重新命名树莓派的主机名。在 raspi-config 的设置页面中，选择“7 Advanced Options”→“A2 Hostname”选项，更改主机名后再重新启动树莓派，就能以新的主机名来进行Bonjour寻址。

Windows系统并没有自带对Bonjour的支持，但是可以通过下载安装iTunes或“Bonjour Print Services for Windows”来获得Bonjour功能。

Bonjour给设备提供了一个动态域名，用于对应该设备的IP地址。在Mac OS X下，可以用下面的命令来查询背后的IP地址：

```
$dns-sd -q raspberrypi.local
```

8.3 互联网SSH登录

介绍了局域网和点对点情况下的SSH登录后，我们可以把野心放大一点，尝试在互联网环境中远程登录SSH，下面用几种不同的方式实现。

1.NAT端口映射

如果我们能拿到树莓派在互联网上的公网IP地址，那么就可以直接用一个命令SSH到该IP地址。问题是，现在大部分局域网都用DHCP来给设备分配网内的私有IP，很可能只有网关才享有一个公网IP地址。有些网关允许设置基于NAT的端口映射。一组公网IP和端口号能对应唯一的私网IP和端口号。在这种情况下，我们就能从外网连接到局域网中的树莓派了，如图8-3所示。

端口映射图



端口映射表			
公网 IP	PORT	内网 IP	PORT
155.66.173.12	12345	192.168.1.1	22

图8-3 端口映射图

我们可以利用这一机制来找到树莓派，比如，通过设置网关，让公网的199.165.145.1:8999对应私网的10.0.0.1:22。这里的199.165.145.1是网关的公网IP。10.0.0.1是树莓派的私网IP。22是SSH协议的默认端口。这时在互联网的其他电脑上，就可以用SSH连接到局域网中的树莓派了：

```
$ssh pi@199.165.145.1:8999
```

为了用该方法，我们的网关必须允许相关的端口映射设置。而很多网关出于安全考虑，完全不向外网开放类似的端口映射。因此，这一方法看似可行，但实践中会遇到很多困难。

2.REMOT3.IT

树莓派官网提供了一种简便的方法，即使用Weaved公司推出的REMOT3.IT。首先要在树莓派上安装相关的工具：

```
$sudo apt-get install weavedconnectd  
$sudo weavedinstaller
```

在安装过程中，REMOT3.IT会要求你输入REMOT3.IT网站的账户信息。在树莓派上安装完成后，在REMOT3.IT网站登录自己的账户，就能看到树莓派设备。如图8-4所示，网站会提供用于在互联网上连接

该树莓派所需的地址和端口号。根据地址和端口号，你就可以在任何
一个能连接到互联网的电脑上，用SSH客户端访问该树莓派。这个服
务很好用，但是该网站不仅限制树莓派的数目，而且限制SSH连接的
时间。想要避免这些限制，就需要缴费了。



图8-4 REMOT3.IT设备管理

3.SSH反向隧道

其实，类似于REMOT3.IT的技术不难自行实现。我们可以用SSH
反向隧道（Reverse Tunneling）技术，从外网远程登录树莓派。首
先，让树莓派主动向公网服务器的某个端口发起SSH连接，比如
vameilab.com:8999，形成一个SSH隧道。当我们使用互联网上的其他
电脑，通过SSH连接到服务器的这一端口时，服务器会把通信内容接
力到与树莓派的SSH隧道中，最终抵达树莓派。整个过程如图8-5所
示。由于公网服务器的域名和IP地址相对固定，因此我们也不用为找
不到树莓派的IP地址而头痛。

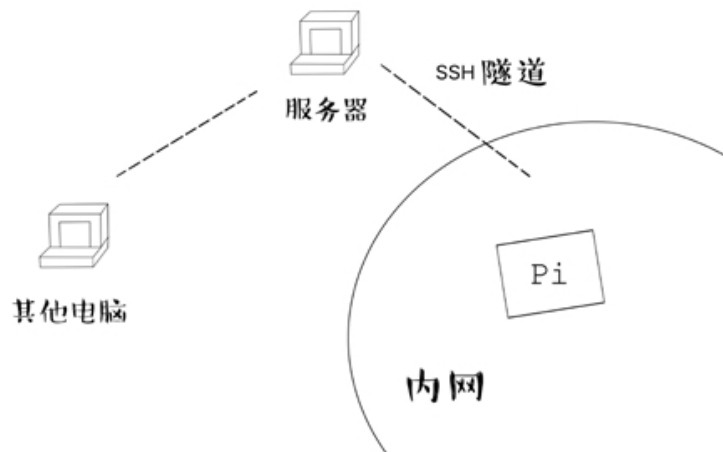


图8-5 SSH反向隧道

了解原理之后，我们也可以自行实现一个类似的中继服务器。你可以使用Amazon或阿里云的弹性云来架设中继服务器。你需要在云的控制台中开放用于反向连接的端口，如8999。从树莓派上用SSH命令建立反向隧道：

```
$ssh -R 8999:localhost:22 vamei@vameilab.com
```

上面的命令，从树莓派的22端口到vameilab.com的8999端口建立反向隧道。登录时用的vamei是中继服务器上的一个账户。反向隧道建立之后，就可以从互联网上直接登录树莓派了：

```
$ssh -p 8999 pi@vameilab.com
```

8.4 文件传输

此前，我们在树莓派上建立了一个SSH服务器，然后通过这个服务器提供的远程Shell来控制树莓派。当需要在本地电脑和树莓派之间传输文件时，我们同样可以利用树莓派上的SSH服务器进行。

1.sftp命令

如果本地电脑是Linux或Mac OS X系统，那么可以使用sftp命令工具来传输文件。首先，在终端下用sftp命令登录树莓派，其登录方式与SSH登录类似：

```
$sftp pi@192.168.1.101
```

输入密码后，sftp会提供一个Shell。你可以用下面的命令来查看树莓派当前目录下的文件：

```
$$ls
```

还可以查看本地电脑当前目录下的文件：

```
$$lls
```

查看树莓派的当前目录：

```
$$pwd
```

更改树莓派上的目录。 ***new_folder*** 是当前目录下的一个子目录：

```
$$cd new_folder
```

返回上级目录使用：

```
$$cd ..
```

查看本地电脑的当前目录：

```
$$lpwd
```

更改本地电脑的目录。 ***new_folder*** 是当前目录下的一个子目录：

```
$$lcd new_folder
```

返回上级目录使用：

```
$$lcd ..
```

从树莓派上下载文件：

```
$$get remote.file
```

文件 **remote.file** 是树莓派当前目录下的一个文件，它将被下载到本地电脑的当前目录下。

把本地电脑的文件上传到树莓派上：

```
$$put local.file
```

文件 **local.file** 是本地电脑当前目录下的一个文件，它将被上传到树莓派的当前目录上。

退出sftp：

```
$$exit
```

2. scp命令

在Linux和Mac OS X下，不仅可以用sftp命令传输文件，还可以用scp命令来传输文件。该命令同样基于SSH，所以树莓派上的SSH服务器要先设置好。这个命令可以在一行命令中完成复杂的功能，避免了手动的Shell操作。我们来看一个文件的上传：

```
$scp local.file pi@192.168.1.101:/home/vamei/
```

这行命令把本地电脑当前目录下的 **local.file** 上传到远程电脑的 **/home/vamei/** 目录。可以看到，scp后面跟着两个参数，分别说明了文件的起始位置和目标位置。在scp中，可以用“用户名@主机：路径”的方式，来指定文件或目录。如果位置在本机上，那么可以省去用户名和主机，只用路径就可以了。

类似的，下载文件：

```
$scp pi@192.168.1.101:/home/vamei/remote.file ./local.file
```

这行命令不但会把树莓派的 **remote.file** 下载到本地电脑的当前目录中，还会把文件重新命名为 **local.file**。因此，目标位置不仅可以是一个目录，还可以是一个新的文件名。

在scp命令中增加-r选项，可以下载整个目录：

```
$scp -r pi@192.168.1.101:/home/vamei .
```

树莓派上的 */home/vamei* 目录会下载到本地电脑的当前目录。命令scp还会遍历 */home/vamei* 下的所有内容，并一一下载下来。上传整个目录的过程与此类似。

3.图形化工具

除了使用命令行来操作树莓派上的文件，我们也可以使用更直观的图形化工具。FileZilla是一款跨平台的FTP/SFTP文件管理工具。从FileZilla的官网下载该软件，FileZilla的软件界面如图8-6所示。它可以用图形化的方式显示出本地电脑和远程树莓派的文件。你可以通过拖曳的方式在两者之间传文件。

单击左上角的服务器图形的按钮可以进入站点编辑器。要想访问树莓派上的文件，我们可以在站点管理器中选择“New Site”选项新建一个站点，填入树莓派的主机名或IP地址，选择SFTP作为“Protocol”。在登录信息中选择Normal作为“Logon Type”，并输入树莓派的登录用户名和密码，如图8-7所示。

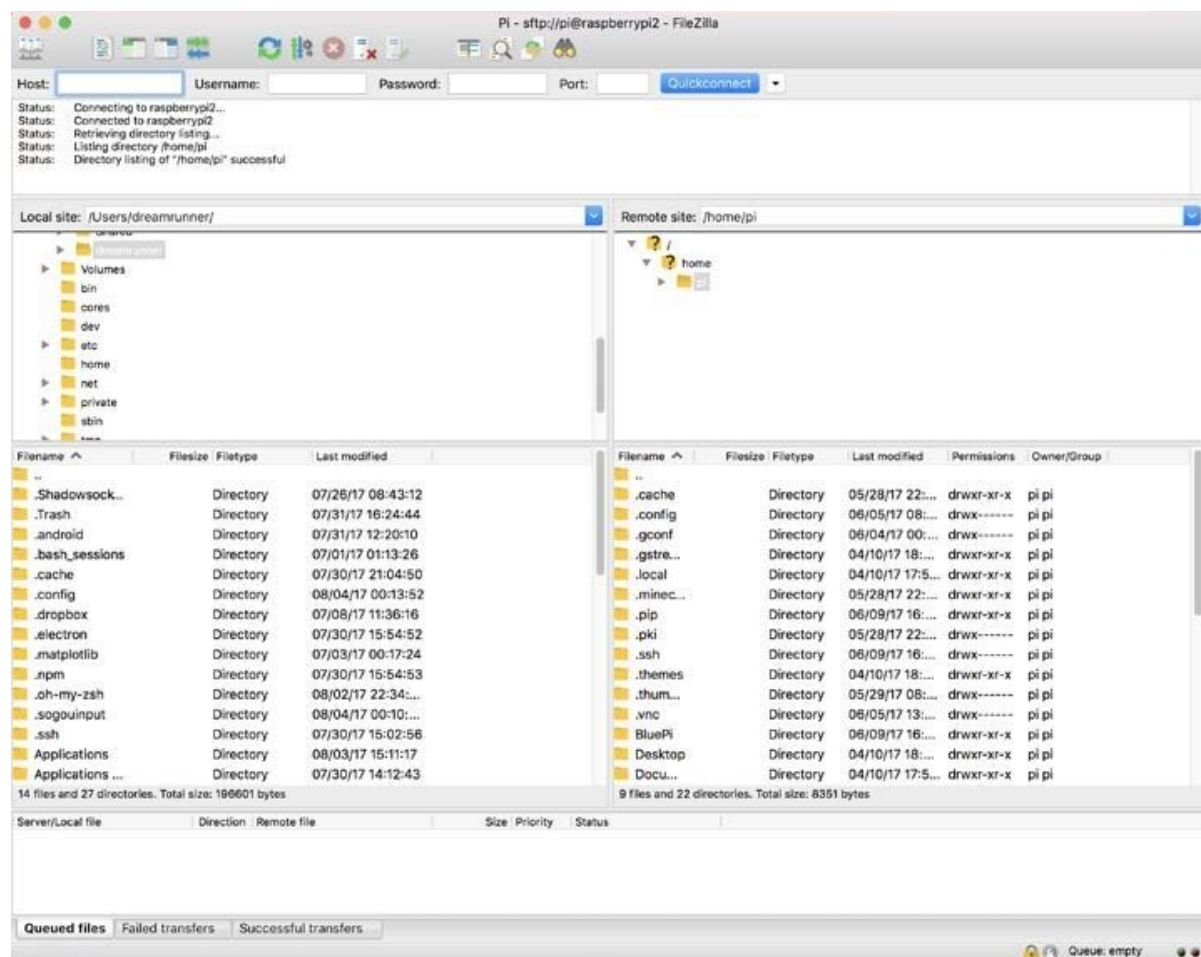


图8-6 FileZilla的软件界面

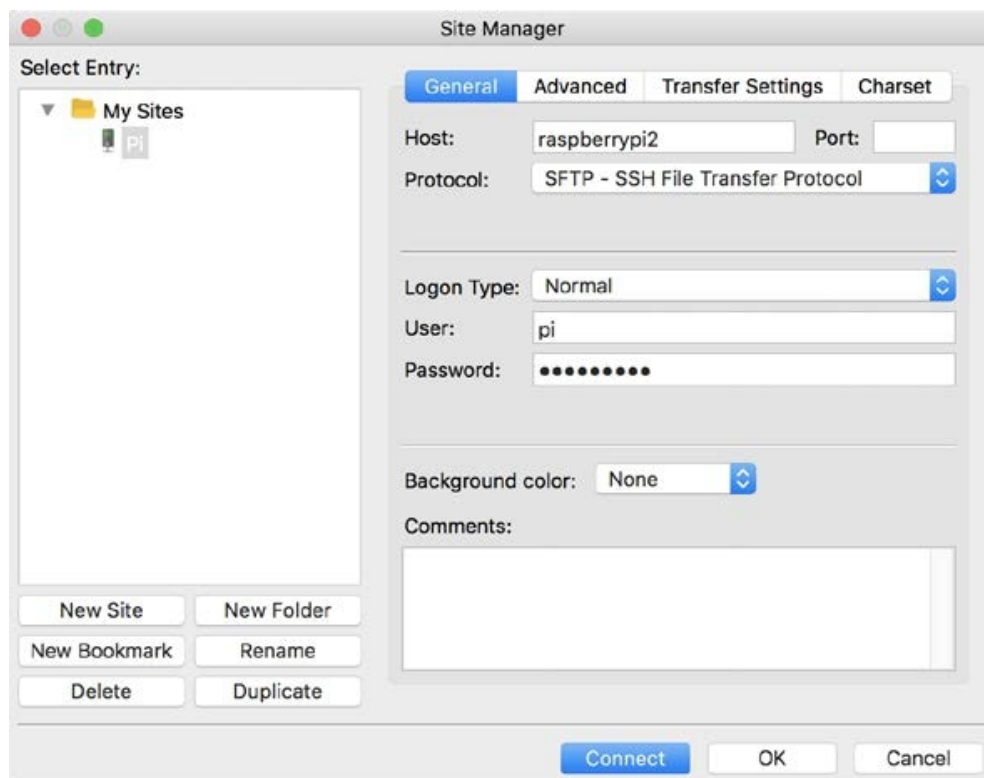


图8-7 站点管理器

第9章 时间的故事

对于电子设备来说，时间都是基础性的功能，很容易被人忽视。20世纪的“千年虫”问题，就是时间方面设计缺陷造成的。对于网络连接的多设备来说，保持时间同步是一个新的问题。对于树莓派的众多应用场景来说，时间的准确性都至关重要。树莓派提供了NTP服务，通过网络来校正时间。即使在断网的情况下，也可以通过物理计时来校正时间。而树莓派使用的Linux系统，也提供了date命令这样便利的时间工具。

9.1 NTP服务

树莓派中内置了NTP服务，所以连上网之后就可以自动调整时间。NTP是**网络时间协议**（Network Time Protocol）的简称，主要用于网络时间的同步。NTP协议早在20世纪80年代就已经诞生了，至今仍是互联网的基础性协议之一。NTP通信分为服务器和客户端两方。客户端发出的数据包包含发出时客户端的时间。服务器收到数据包并回复。在回复的数据包中，附加了服务器收到和发出数据包的时间。客户端收到回复后就可以获得网络延迟时间，以及自己和服务器的时间差。客户端据此调整自己的时钟，就可以与服务器时间保持同步。

你可以通过下面的命令来查询当前使用的NTP服务器：

```
$sudo ntpq -pn
```

命令返回：

remote	refid	st	t	when	poll	reach	delay	offset	jitter
203.135.184.123	.GPS.	1	u	322	64	20	365.136	-7.571	15.792
223.112.179.133	.INIT.	16	u	-	1024	0	0.000	0.000	0.000
*202.112.29.82	202.118.1.46	2	u	122	64	276	53.148	0.766	0.868

行首加*号的是当前服务器。此外，还列出了网络**延迟时间**（Delay）、与服务器时间差（Offset）等关键的NTP时间数据。单位是**毫秒**（Millisecond）。

如果NTP服务出现问题，就会造成树莓派时间错误，可以强制要求NTP对表：

```
$sudo service ntp stop
$sudo ntpd -gq
$sudo service ntp start
```

上面的第一句和第三句分别用于停止和启动NTP服务。

即使不使用NTP，也可以手动调整系统时间：

```
$sudo date -s "1 Jan 2017 00:00:00"
```

即把系统时间调整为2017年1月1日00:00:00。

然后用date命令来显示系统当前时间：

```
$date
```

可以看到，时间已经调整成功。

9.2 时区设置

因为地球自西向东转动，所以全球不同经度地点的日出、日落及正午的时间不同。人们又习惯用同样的12点来代表正午，这意味着不同经度的人要用不一样的表。可是，如果每时每刻都要根据经度调表，就会非常麻烦。因此，地球以15°的经度来划分时区，一个时区内用统一的时间，向东跨过一个时区，就需要把表调快1小时。当然，时区不是严格按照15°划分的。比如，一些地跨多个时区的国家有可能统一用一个时区，例如中国。

对于不同地区的用户来说，往往需要把树莓派调整成当地的时区。你可以用raspi-config进入树莓派的设置页面，在“4 Localisation Options”→“I2 Change Timezone”页面中修改时区。

当然，也可以用下面的命令手动修改：

```
$sudo cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
```

目录 `/usr/share/zoneinfo` 中有多个以各大洲名字命名的文件夹，里面的文件以该州的主要城市命名。把对应城市的文件复制到 `/etc/localtime`，就可以把系统的时区设成该城市所用的时区。这里把时区修改为“Shanghai”，也就是上海。修改之后，用date命令查看时间，可以看到时区简写变成CST，也就是“上海时间”的缩写：

```
Tue 3 Jan 20:42:24 CST 2017
```

用date命令查看UTC时间：

```
$date -u
```

显示的时间正好相差8个小时：

```
Tue 3 Jan 12:42:24 UTC 2017
```

9.3 实时时钟

大多数电脑的主板上包含了一个**实时时钟**（RTC，Real Time Clock）。实时时钟是一个有电源的表，能在电脑断电时继续计时。因此，电脑断电后一天再开机，你会发现电脑的时钟也往前走了一天。树莓派并不包含一个实时时钟，因此，如果树莓派断电一天再开机，在NTP服务校正时间之前，树莓派的时间还停留在一天前。为了解决这一问题，可以给树莓派附加一个实时时钟，比如PiFace专门为树莓派设计的实时时钟。

这个实时时钟被设计成一个使用纽扣电池的电路板。把PiFace电路板的孔对准树莓派的GPIO针脚插入，就可以使用了。插入位置如图9-1所示。插入正确的情况下，电池正好在树莓派CPU的上方。网上也有人诟病这一设计，认为电池的发热会影响树莓派CPU的散热。不过笔者在使用中并没有遇到太大问题。

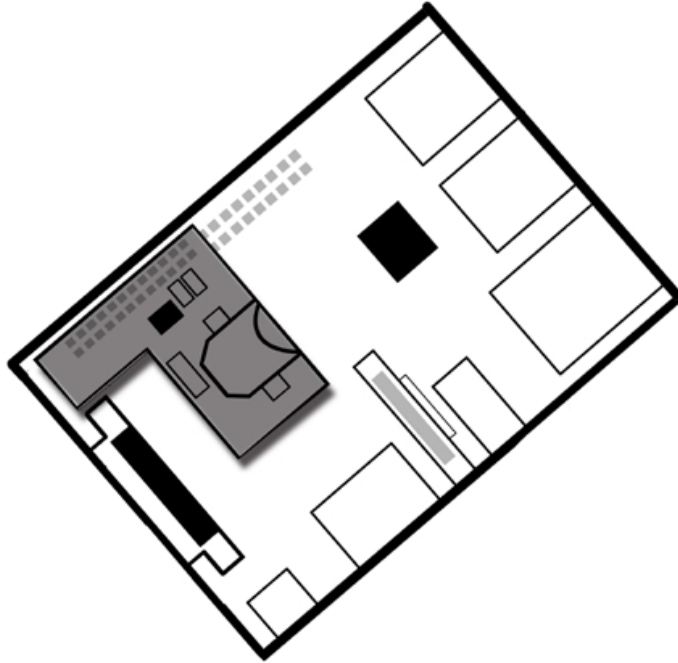


图9-1 RTC安装位置

为了使用这款实时时钟，还需要进行一些设置。首先，这块电路板是通过I2C接口与树莓派通信的，所以要在raspi-config的页面中打开I2C接口。然后，安装所需的工具包：

```
$sudo apt-get install i2c-tools  
$sudo apt-get install python-smbus
```

接下来，赋予用户pi使用I2C接口的权限：

```
$sudo usermod -aG i2c pi
```

打开文件 **/etc/modules**，这里面列出了系统可以加载的模块。检查是否有如下两行，如果没有请添加：

```
i2c-dev  
i2c-bcm2708
```

下面一段程序修改自官网程序，用来让树莓派在开机时自动加载实时时钟。把下面程序保存为 **rtc.bash**，并运行：

```
#!/bin/bash
```

```

#=====
# NAME: set_revision_var
# DESCRIPTION: Stores the revision number of this Raspberry Pi into
#              $RPI_REVISION
#=====

set_revision_var() {
    revision=$(grep "Revision" /proc/cpuinfo | sed -e "s/Revision\t: //" )
    RPI2_REVISION=$((16#a01041))
    RPI3_REVISION=$((16#a02082))
    if [ "$((16#$revision))" -ge "$RPI3_REVISION" ]; then
        RPI_REVISION="3"
    elif [ "$((16#$revision))" -ge "$RPI2_REVISION" ]; then
        RPI_REVISION="2"
    else
        RPI_REVISION="1"
    fi
}

#=====
# NAME: start_on_boot
# DESCRIPTION: Load the I2C modules and send magic number to RTC, on boot.
#=====
start_on_boot() {
    echo "[info]Create a new pifacertc init script to load time from PiFace
RTC."
    echo "[info]Adding /etc/init.d/pifacertc ."
}

```

```

if [[ $RPI_REVISION == "3" ]]; then
    i=1 # i2c-1
elif [[ $RPI_REVISION == "2" ]]; then
    i=1 # i2c-1
else
    i=0 # i2c-0
fi

cat > /etc/init.d/pifacertc << EOF
#!/bin/sh
### BEGIN INIT INFO
# Provides:          pifacertc
# Required-Start:    udev mountkernfs \$remote_fs raspi-config
# Required-Stop:
# Default-Start:     S
# Default-Stop:
# Short-Description: Add the PiFace RTC
# Description:       Add the PiFace RTC
### END INIT INFO

. /lib/lsb/init-functions

case "$1" in
start)
    log_success_msg "Probe the i2c-dev"
    modprobe i2c-dev
    # Calibrate the clock (default: 0x47). See datasheet for MCP7940N
    log_success_msg "Calibrate the clock"
    i2cset -y $i 0x6f 0x08 0x47
    log_success_msg "Probe the mcp7941x driver"
    modprobe i2c:mcp7941x
    log_success_msg "Add the mcp7941x device in the sys filesystem"
    # https://www.kernel.org/doc/Documentation/i2c/instantiating-devices
    echo mcp7941x 0x6f > /sys/class/i2c-dev/i2c-$i/device/new_device
    log_success_msg "Synchronise the system clock and hardware RTC"
    hwclock --hctosys
    ;;
stop)
    ;;
restart)
    ;;
force-reload)
    ;;
*)
    echo "Usage: \`$0 start\` >&2

```

```

        exit 3
    ;;
esac
EOF
    chmod +x /etc/init.d/pifacertc

    echo "[info]Install the pifacertc init script"
    update-rc.d pifacertc defaults
}

set_revision_var &&
start_on_boot

```

完成后重启电脑。此时树莓派应该已经自动通过I2C接口加载了实时时钟。可以通过下面的命令来检查实时时钟是否就位：

```
$sudo i2cdetect -y 1
```

如果实时时钟就位，那么60开头的行会有一个“UU”的标准位。通过下面的命令，可以读出实时时钟的时间：

```
$sudo hwclock -r
```

通过下面的命令可以把当前系统时间写入实时时钟：

```
$sudo hwclock --systohc
```

有了实时时钟，就可以在无网环境下保持时间的连续性了。PiFace的产品卖得有些贵，淘宝上有一些便宜的实时时钟可以选购。

9.4 date的用法

date是UNIX系统下常用的时间命令工具，能提供非常丰富的时间功能，比如以特定格式显示时间：

```
$date +"%Y year %m month %d day"
```


+号后面的字符串代表了时间显示格式，%开头的标识符会用时间信息填充。%Y代表年，%m代表月份，%d代表日期，所以上面的命令会返回：

```
2017 year 01 month 01 day
```

用于控制date输出格式的标识符还有很多：

- %a，显示一周中星期几的缩写，比如Thu。
- %A，显示一周中星期几，比如Thursday。
- %b，显示月份的缩写，比如Feb。
- %B，显示月份，比如February。
- %d，显示一个月的哪一天，比如09。
- %D，显示日期，月/日/年，比如02/07/17。
- %F，显示日期，年-月-日，比如2017-02-07。
- %H，显示24小时制的小时，比如23。
- %I，显示12小时制的小时，比如11。
- %j，显示一年中的天数，比如138。
- %m，显示月份，比如02。
- %M，显示分钟，比如14。
- %S，显示秒，比如47。
- %N，显示纳秒，比如264587606。
- %T，显示24小时制的时间，时:分:秒，比如23:44:17。
- %u，显示一周中的哪一天，周一为1。
- %U，显示一年中的周数，而周日是一周的开始，比如47。
- %Y，显示完整的年份，比如2013。
- %Z，显示时区的缩写。

除了显示当前时间，date还可以用来显示用户输入的时间：

```
$date --date="2017/01/03 12:00:00"
```

配合上面介绍的格式控制，这个功能可以实现日期格式的转换。这个功能还可以用于时间推算。比如下面的命令就可以用于推算2016年11月12日之前1个月的时间：

```
$date --date="2016/11/12 -1 month"
```

除了“-1 month”，还可以是“+1 second”“-2 day”等多种时间差，能满足各种各样的时间推算需求。

第10章 规划小能手

树莓派是一款低成本的电脑，因此它常充当小型的服务器，定期执行某些任务。笔者平时就会在局域网下接入树莓派，做一些数据备份和上传的工作。这时任务内容和执行时间已经明确。我们想把任务内容和执行时间预先写入树莓派中，让树莓派自动执行。这样用户就不用手动操作树莓派了。为了满足这一需求，Linux系统提供了经典的cron工具。

10.1 用cron规划任务

cron是Linux系统下常用的任务规划软件，可以在cron中要求系统在特定的时间执行特定的任务。cron在系统中有一个运行着的守护进程。当系统时间符合某一条规划记录时，守护进程就会启动相应的任务。在树莓派命令行中运行下面的命令，就可以找到cron的守护进程：

```
$ps aux | grep cron
```

这一行实际上调用了两个命令：用于查询进程的ps命令和用于文本搜索的grep命令。其中的|是管道符号，它像管道一样，把ps命令的输出传给grep命令作为输入。

结果如下：

```
root      424  0.0  0.2  5072  2384 ?        Ss   14:40   0:00 /usr/sbin/cron -f
pi        6938  0.0  0.2  4280  2008 pts/1    S+   17:42   0:00 grep --color=auto
cron
```

记录中的第一条就是cron的进程。

如果想要规划任务，那么可以用下面的命令来编辑规划记录：

```
$crontab -e
```

在规划记录中，每一行为一条记录，以#开始的是注释。每一行记录又分为6列，用空格分隔，分别表示分钟（m，0~59）、小时（h，0~23）、一个月中的哪一天（dom，1~31）、月（mon，1~12）、一个星期中的哪一天（dow，0~6），以及要执行的命令。在填写规划时间时，除了用数字，还可以用*表示所有：

```
# m h dom mon dow  command
30 5  10  3  *   touch /tmp/test.log
```

上面表示每年3月10日5点30分，执行touch命令。

```
# m h dom mon dow  command
10 18 * * *   echo "Hello World" > /home/pi/log
```

上面表示每天的18点10分执行echo命令。

在同一列中，还可以规划多个时间点，例如：

```
# m h dom mon dow  command
10 2-4 * * *   echo "Hello World" > /home/pi/log
```

每天2:10、3:10和4:10执行。也就是说，“2-4”表示了从2到4的范围。

```
# m h dom mon dow  command
30 1,5 * * *   echo "Hello World" > /home/pi/log
```

每天1:30和5:30执行。也就是说，“1,5”表示了1和5两个时间点。

规划记录crontab保存后，cron就将按照规划，在对应的时间执行对应的命令。每个用户有一个自己的crontab，当cron要执行规划时，也会以相应的用户身份来执行。这里是以pi用户修改保存的crontab，cron就会以pi的身份来运行各个命令。如果想修改其他用户的crontab，那么可以用-u关键字：

```
$sudo crontab -e -u root
```

10.2 用cron开机启动

cron除了做时间规划，还可以用于开机启动。在crontab中添加下面一行记录，就可以方便地实现开机启动：

```
@reboot touch /home/pi/reboot.log
```

10.3 用/etc/init.d实现开机启动

树莓派的 **/etc/init.d** 文件夹下有很多脚本，比如cron。cron脚本把cron这个守护进程包装成了一个服务，定义了它在启动、重启和终止时的具体行为。这样，用户在启用相应服务时，就不用进行太复杂的设置。当服务终止时，操作系统也能根据脚本的定义，自动回收相关资源。用户还能把重要的服务设置成开机启动，省去了手动开启的麻烦。因此，可以在 **/etc/init.d** 中看到很多默默工作的服务，如ssh、bluetooth、rsync等。

服务脚本遵循特定的格式，如下面的 **/etc/init.d/test** 脚本：

```
#!/bin/sh
# Start/stop the test daemon.
#
### BEGIN INIT INFO
# Provides:          test
# Required-Start:    $remote_fs $syslog $time
# Required-Stop:     $remote_fs $syslog $time
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: test
# Description:       test
### END INIT INFO

do_start() {
    echo "start"
}

do_stop() {
    echo "stop"
}

do_restart() {
    echo "restart"
}

do_status() {
    echo "status"
}

do_fallback() {
    echo "fallback"
}

case "$1" in
start) do_start
      ;;
stop)  do_stop
      ;;

```

```
restart) do_restart
        ;;
status) do_status
        ;;
*)      do_fallback
        ;;
esac
exit 0
```

脚本的一开始有头部信息。头部信息中除了基本的介绍，还有其他信息。Required-Start说明了该test应用启动前，系统必须启动的其他应用。Required-Stop列出的应用必须在test应用结束后结束。Default-Start和Default-Stop中说明了默认运行级别。Linux系统可以在不同运行模式下工作，如单用户模式、多用户模式，每种模式就称为一个运行级别。Linux系统中运行级别的意义如下：

- 0 停机，关机。
- 1 单用户，无网络连接，不运行守护进程，不允许非超级用户登录。
- 2 多用户，无网络连接，不运行守护进程。
- 3 多用户，正常启动系统。
- 4 用户自定义。
- 5 多用户，带图形界面。
- 6 重启。

test脚本中，默认支持的运行级别是2、3、4、5。

在脚本的主体程序中包含了一个case分支结构，说明了应用在进入**启动**（start）、**停止**（stop）、**重启**（restart）、**状态查询**（status）状态时应该采用的动作。我们可以用service命令手动让脚本切换状态：

```
$sudo service test start
```

脚本中相应的动作会被调用。

/etc/init.d/myscript 还不能随开机启动。Linux在开机启动时，真正检查的是 **/etc/rcN.d** 文件夹，执行其中的脚本。这里的N代表了运行级别。

比如说在运行级别2时，Linux会检查 */etc/rc2.d* 文件夹，执行其中的脚本。我们需要把 */etc/init.d* 中的服务复制到或者建立软链接到 */etc/rcN.d* 上，才能让该服务在N运行级别开机时启动。不过，我们可以利用update-rc.d命令更方便地进行，比如在默认的运行级别建立软链接：

```
$sudo update-rc.d cron defaults
```

以及删除默认运行级别下的软链接：

```
$sudo update-rc.d cron remove
```

10.4 避免使用*/etc/rc.local*

树莓派官网上给出了修改 */etc/rc.local* 的方法，以便在树莓派开机时执行用户自定义的任务。比如在该文件中执行date命令：

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.

# time
date > /tmp/rc.local.log

exit 0
```

但笔者不推荐这种启动方式。*/etc/rc.local* 是在系统初始化的末尾执行的一个脚本。如果把太多的任务加入这个脚本中，不但会拖慢开机速度，还会造成管理上的混乱。因此，*/etc/rc.local* 往往只用于修改一些在

启动过程需要设定的参数，而不涉及具体的任务启动。如果想随开机启动某些服务，应该避免使用 */etc/rc.local* 。

10.5 Shell中的定时功能

很多命令自身也带有定时功能，比如关机命令shutdown：

```
$sudo shutdown +10
```

即10分钟后关机。

说明关机的时间：

```
$sudo shutdown 22:12
```

还可以使用sleep命令，让Shell等待一段时间：

```
$sleep 10 && echo hello
```

这里的&&符号连接了两个命令。对于&&符号连接的两个命令，bash会在第一个命令执行成功后才执行第二个。由于第一个命令是让Shell等待10秒，因此输入这行命令后，Shell会在10秒后执行echo hello命令。

第11章 GPIO的触手

树莓派可以通过很多接口来连接到其他设备。在各种各样的接口中，最有特色的就是一组GPIO（General Purpose Input/Output）接口。这组GPIO接口大大拓展了树莓派的能力。GPIO不仅能实现通信，还能直接控制电子元器件，从而让用户体验到硬件编程的乐趣。

11.1 GPIO简介

树莓派3上的GPIO接口由40个针脚（PIN）组成，如图11-1所示。

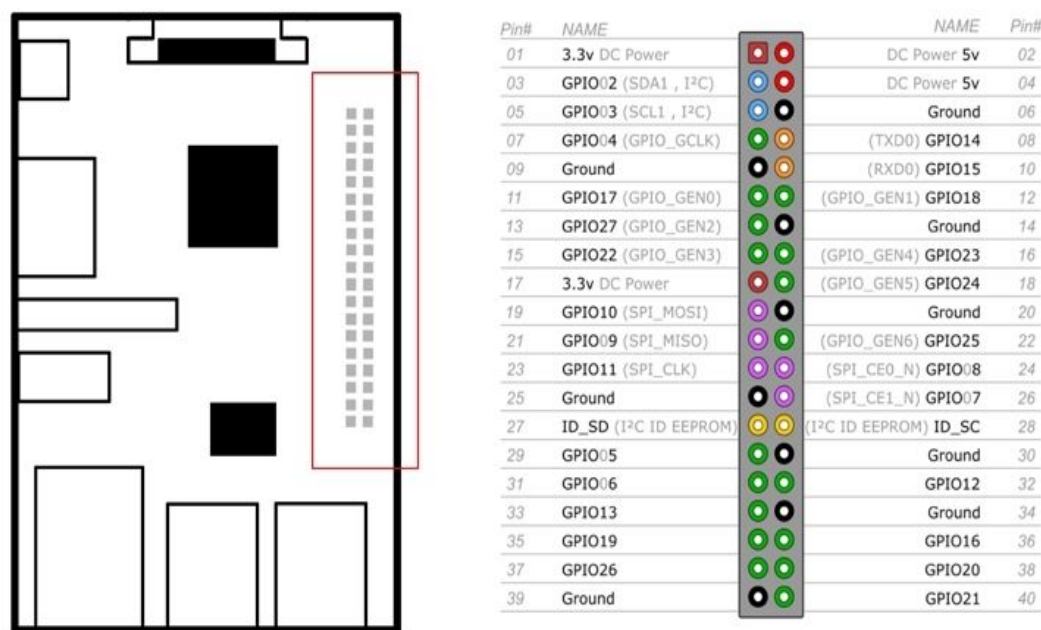


图11-1 树莓派3的GPIO针脚

每个针脚都可以用导线和外部设备相连。你可以通过焊接的方式把导线固定在PIN上，也可以用母型的跳线套接在PIN上。在40个PIN中，有固定输出的5V（2、4号PIN）、3.3V（1、17号PIN）和地线（Ground，6、9、14、20、25、30、34、39）。如果一个电路两端接在5V和地线之间，那么该电路就会获得5V的电压输入。27和28号PIN标着ID_SD和

ID_SC。它们是两个特殊的PIN，用于和附加的电路板通信。其他的PIN大多编成GPIOX的编号，如GPIO14。树莓派的操作系统会用GPIO的编号14来指代这个PIN，而不是位置编号的8。

有一些PIN不仅有GPIO功能，还能充当其他形式的端口。比如，GPIO14和GPIO15除了作为GPIO接口，还可以充当UART端口。此外，GPIO上还能找到I2C和SPI接口。

计算机中用高、低两个电压来表示二进制的1和0。树莓派上的GPIO用相同的方式来表示数据。每个GPIO的PIN都能处于输入或输出状态。当处于输出状态时，系统可以把1或0传给该PIN。如果是1，那么对应的物理PIN向外输出3.3V的高电压，否则输出0V的低电压。相应的，处于输入状态的PIN可以探测物理PIN上的电压。如果是高电压，那么该PIN将向系统返回1，否则返回0。利用简单的二元机制树莓派实现了和物理电路的互动。

11.2 控制LED灯

我们先来看GPIO输出的一个例子。在GPIO21和地线之间接一个串联电路，电路上有一个LED灯，还有一个用于防止短路的330Ω电阻。当GPIO21位于高电平时，将有电流通过电路点亮LED灯，如图11-2所示。

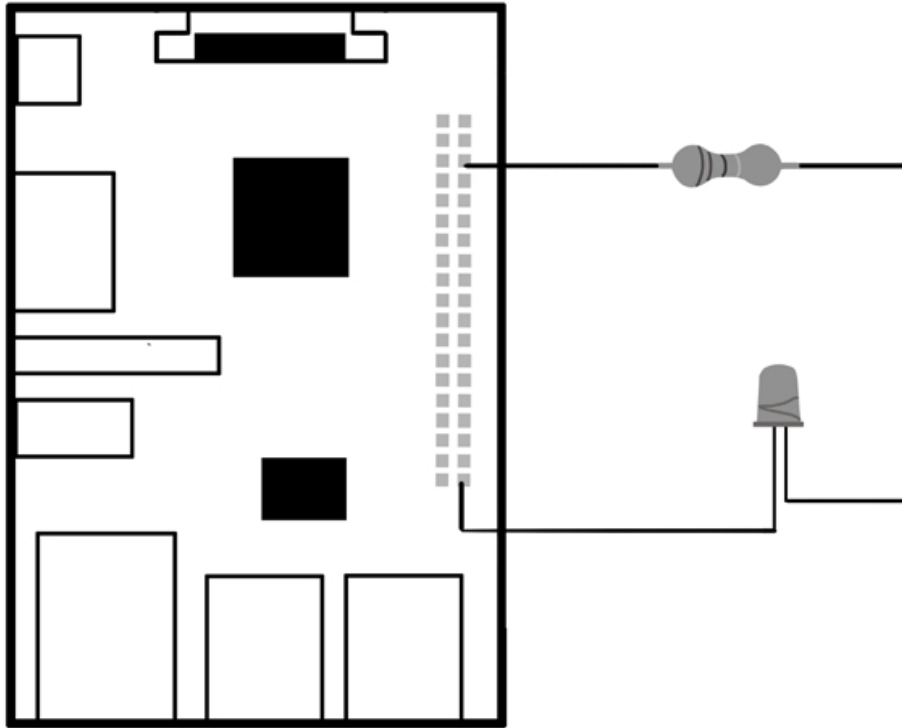


图11-2 GPIO与LED灯连接

我们通过Shell来控制GPIO21。在Linux中，外部设备经常被表示成文件。向文件写入或读取字符，就相当于向设备输出或者从设备输入字符。树莓派上的GPIO端口也是如此，其代表文件位于 */sys/class/gpio/* 下。

首先，激活GPIO21：

```
$echo 21 > /sys/class/gpio/export
```

这个命令把字符“21”输入 */sys/class/gpio/export* 中。命令执行后，*/sys/class/gpio/* 下面增加了代表GPIO21的一个目录，目录名就是gpio21。

其次，把GPIO21置于输出状态：

```
$echo out > /sys/class/gpio/gpio21/direction
```

文件 */sys/class/gpio/gpio21/direction* 用于控制GPIO21的方向，向里面写入了代表输出的字符“out”。

最后，向GPIO21写入1，从而让PIN处于高电压：

```
$echo 1 > /sys/class/gpio/gpio21/value
```

可以看到，LED灯亮了起来。

如果想关掉LED灯，那么只需要向GPIO21写入0：

```
$echo 0 > /sys/class/gpio/gpio21/value
```

使用完GPIO21可以删除该端口：

```
$echo 21 > /sys/class/gpio/unexport
```

/sys/class/gpio/gpio21 随即消失。

11.3 两个树莓派之间的GPIO

我们可以用GPIO的方式连接两个树莓派，如图11-3所示。一个树莓派的GPIO输出将成为另一个树莓派的GPIO输入。连接方式很简单，只需要两根导线。一根导线连接两个树莓派的地线，另一根导线连接树莓派的两个PIN。

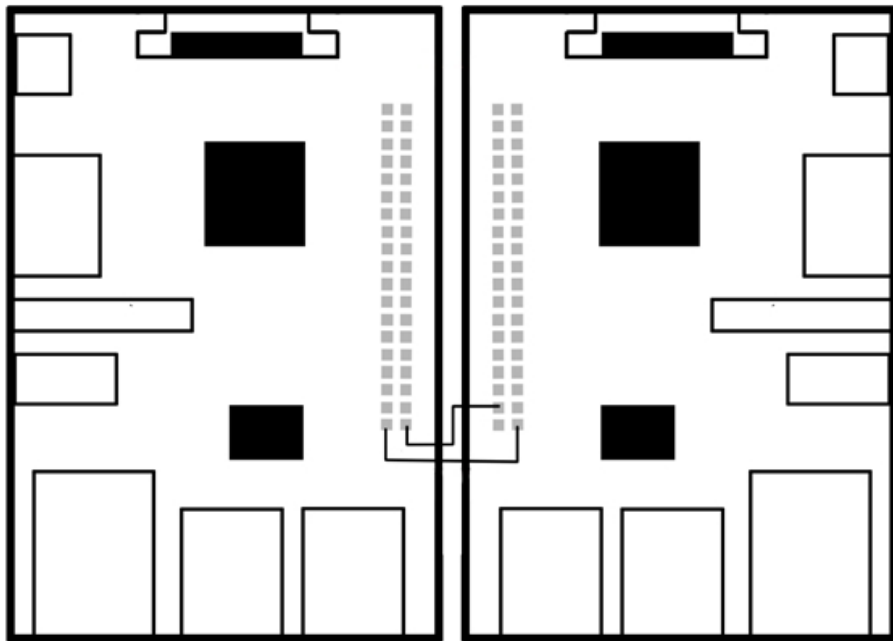


图11-3 两个树莓派之间用GPIO连接

用左侧的树莓派来输出，用右侧树莓派来输入。输出过程和上面控制LED灯的例子相似。第一个树莓派中的GPIO21准备输出：

```
$echo 21 > /sys/class/gpio/export
$echo out > /sys/class/gpio/gpio21/direction
```

在第二个树莓派中，准备好读取GPIO26：

```
$echo 26 > /sys/class/gpio/export
$echo in > /sys/class/gpio/gpio26/direction
```

当我们向 `/sys/class/gpio/gpio26` 中写入“in”时，就是把GPIO26置于输入状态。

此后，在第一个树莓派中就可以更改输出值为1或0了：

```
$echo 1 > /sys/class/gpio/gpio21/value
$echo 0 > /sys/class/gpio/gpio21/value
```

在第二个树莓派中，可以用cat命令来读取文件，获得输入值：

```
$cat /sys/class/gpio/gpio26/value
```

cat命令读完一次后会返回，为了持续读取，可以用bash中的while循环来反复调用cat：

```
$while true; do cat /sys/class/gpio/gpio26/value; done
```

这里的while是bash提供的循环结构，随后do和done之间的内容会重复执行。第二个树莓派将循环查看GPIO26的输入值。当第一个树莓派中的输出改变时，第二个树莓派获得的输入也随之改变。我们在两个树莓派之间实现了简单的通信。

最后，在使用完GPIO后，别忘了删除端口。

11.4 UART编程

计算机的数据是许多位的0和1构成的序列。尽管GPIO可以在0和1之间切换，但无法传输二进制序列。比如，把一个二进制序列11000111输出到GPIO端口，在输入端看来，只是输入了一段时间的1，然后变成0，最后又变成1。输入端无法准确说出，一段高电平输入究竟包含了几位1。

一个解决方案是用多个PIN同时通信，每个PIN表示一位。当输入端读取完成后，通知输出端，让输出端送来下一批的数据。这种通信方式被称为并口传输。和并口传输对应的是串口传输，传输时依然是用一个PIN，但输入方可以知道一位数据持续了多长时间。GPIO上的UART、I2C、SPI都是串口通信。

UART与其余两者的区别在于，通信双方通过事先约定的速率来发送或接收数据。这种通信方式称为异步通信。I2C和SPI这类同步通信方式会用额外的连线来保证双方速率相同。UART的连线和实现方式很简单，因而成为最流行的串口通信方式。但UART的缺点在于，如果发送方和接收方的速率不同，那么通信就会发生错误。通信速率称为**波特率**（Baudrate），单位是每秒通信的**位数**（bps）。

UART的端口至少有RX、TX和地线三个针脚。RX负责读取，TX负责输出。如果有两个UART端口，它们的连接方式如图11-4所示。

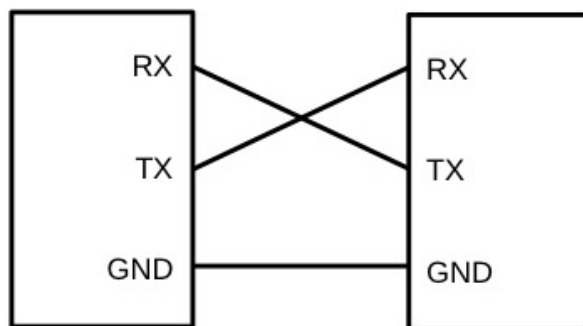


图11-4 UART连接

在树莓派3上，TX和RX就是GPIO14和GPIO15针脚。因此，我们可以把两个树莓派按照图11-4的方式连接起来，然后在两个树莓派之间实现UART通信。

在这里，我们要注意树莓派3发生了一点变化。树莓派1和2中都使用了标准的UART，在操作系统中的对应文件是 `/dev/ttyAMA0`。在树莓派3中，新增的蓝牙模块占用了标准UART端口和树莓派沟通，外部的UART通信采用了简单的Mini UART，在操作系统中的对应文件是 `/dev/ttyS0`。

由于mini UART的波特率依赖于CPU时钟频率，而CPU时钟频率可能在运行过程中浮动，因此mini UART经常会带来意想不到的错误。一般有两种解决方案：一种是关闭蓝牙模块，让外部连接重新使用标准UART端口；另一种是固定CPU时钟频率，以便mini UART能以准确的波特率进行通信。

关闭蓝牙模块，需要修改 */boot/config.txt* 。将dtooverlay键的值改为：

```
dtooverlay=pi3-disable-bt
```

修改后重启。此后的UART通信，就可以通过 */dev/ttyAMA0* 进行。

如果采取第二种解决方案，那么要修改 */boot/config.txt* ，把上面的修改变成：

```
core_freq=250  
dtooverlay=pi3-miniuart-bt
```

修改后重启。此后的UART通信就可以通过 */dev/ttyS0* 进行了。

我们以第一种解决方案为例，进行UART通信。首先，设定波特率：

```
$stty -F /dev/ttyAMA0 9600
```

然后，向UART端口输出文本：

```
$echo "hello" > /dev/ttyAMA0
```

在UART的另一端读取文本：

```
$cat /dev/ttyAMA0
```

可以看到，UART可以实现更加复杂的文本通信。如果使用第二种解决方案，即限定核心频率的办法，那么只需把 */dev/ttyAMA0* 改为 */dev/ttyS0* 即可。

11.5 用UART连接PC

一般的PC都没有暴露在外的UART针脚。为了通过UART来连接PC和树莓派，我们需要一个USB和UART的转换器。这个转换器的一端是USB接口，另一端是UART的针脚。我们把USB一端插入PC，另一端按照UART到UART的方式，连接到树莓派的UART针脚。

连接好之后，就可以在PC上利用串口操作软件来和树莓派通信了。在Linux下，USB连接表示为 `/dev/ttyUSB0`。当然，当计算机上只有1个USB设备时，最后的编号才会是0。而在笔者的Mac OS X上，该USB连接被表示成 `/dev/cu.SLAB_USBtoUART`。此后，就可以通过操作USB文件来进行UART通信了。

11.6 用UART登录树莓派

我们还可以用UART的方式连接并登录树莓派。进入树莓派设置：

```
$sudo raspi-config
```

在Interfacing Options → Serial中，允许开机时通过串口登录。

重启后，树莓派启动时会自动把开机信息以115200的波特率推到UART端口。在UART另一端的PC上，如果使用Mac OS X，那么可以用下面的命令连接：

```
$screen /dev/cu.SLAB_USBtoUART 115200
```

如果PC是Linux系统，则只需把USB设备文件改为对应的设备文件即可。如果是Windows系统，则可以使用Putty通过串口连接树莓派。首先在Windows的设备管理器中找到该USB设备。假如USB设备被识别为COM3，那么在Putty的设置页面中，把**连接类型**（Connection Type）设置成**串口**（Serial），然后在**串口线路**（Serial Line）中输入USB设备的名称，例如COM3。**速度**（Speed）选择115200。设置好后单击“**打开**（Open）”按钮即可连接。

第12章 玩转蓝牙

蓝牙是一个使用广泛的无线通信协议，这两年又随着物联网概念进一步推广。本章介绍蓝牙协议，特别是低功耗蓝牙，并用树莓派来实践。树莓派3中内置了蓝牙模块。树莓派通过UART接口和该模块通信。树莓派1和树莓派2中没有内置的蓝牙模块，不过你可以通过USB安装额外的蓝牙适配器。本章以树莓派3为基础，介绍蓝牙通信。

12.1 蓝牙介绍

蓝牙由爱立信创制，旨在实现不同设备之间的无线连接。蓝牙无线通信的频率为2.4GHz，和Wi-Fi一样，都属于特高频。相对于低频信号来说，高频传输的速度比较快，穿透能力强，但传输距离受限。在没有遮蔽和干扰的情况下，蓝牙设备的最大通信距离能达到30米。但大多数情况下，蓝牙的实际通信距离在2到5米。相比之下，使用低频433MHz的对讲机设备，其通信距离很容易超过百米。因此，蓝牙常用于近距离的无线设备，比如无线鼠标和键盘。蓝牙的标志如图12-1所示。蓝牙的工作流程可以分为下面三个基本步骤。



图12-1 蓝牙的标志

- 广播/扫描：通信的一方向外广播自己的信息，另一方通过扫描知道自己周边有哪些蓝牙设备在广播，这些设备的地址是什么，以及是否

可以连接，如图12-2所示。

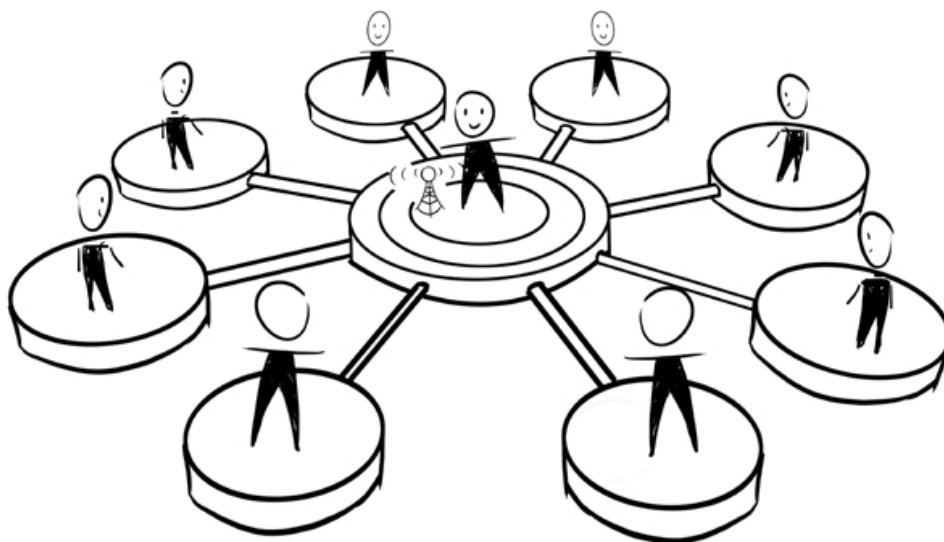


图12-2 广播

· 连接：通信的一方向另一方发起连接请求，双方通过一系列的数据交换建立连接，如图12-3所示。

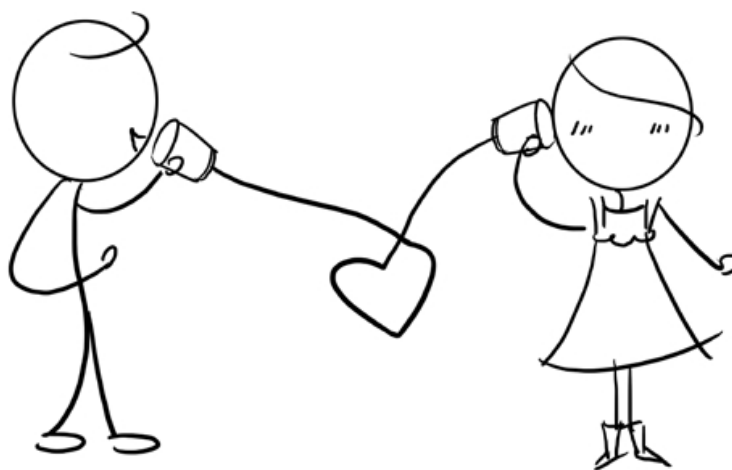


图12-3 连接

· 数据通信。

根据细节上的差别，蓝牙通信又细分为两种：经典蓝牙和低功耗蓝牙。早期的蓝牙通信方式称为**经典蓝牙**（Classic Bluetooth）。经典蓝牙中的数据通信协议是串行仿真协议RFCOMM。RFCOMM仿真了常见的串

口连接。数据从一端输入，从另一端取出。经典蓝牙的开发非常简单。基于串口开发的有线鼠标程序，就可以直接用于RFCOMM连接的无线鼠标程序。此外，经典蓝牙可以快速传输数据。因此，诺基亚N95这样的早期智能手机，也用RFCOMM来互传图片和文件。

12.2 BLE介绍

经典蓝牙的缺点是比较耗电。后来，诺基亚发明了一种可以降低功耗的蓝牙通信方式。2010年出台的蓝牙4.0把这种通信方式规范为“**低功耗蓝牙**”（BLE，Bluetooth Low Energy）。BLE把通信双方分为非对称的双方，尽量让其中的一方承担主要的开销，减轻另一方的负担。举例来说，当手环与手机通信时，手环电量少，而且需要长时间待机。BLE通信的主要负担可以放在电量较充裕且充电方便的手机一侧，从而减少手环的能耗。

BLE通信一般也包含广播/扫描的步骤。主动发起广播的设备称为**外设**（Peripheral），扫描设备称为**中心设备**（Central）。BLE连接成功之后，就可以开始数据传输。BLE的数据传输协议是ATT协议和GATT协议。ATT是GATT的基础。ATT协议把通信双方分为**服务器**（Server）和**客户端**（Client）。客户端主动向服务器发起读写操作。需要注意的是，ATT中的服务器和客户端，与广播阶段的外设和中心设备相互独立。当然，在手环这样的应用场景下，外设通常也是服务器。ATT协议以**属性**（Attribute）为单位进行该数据传输。一个属性的格式有以下四个部分：

handle	type	value	permission
--------	------	-------	------------

我们分别来理解属性的不同部分。

- handle：句柄，包括了属性的唯一编号，长度为16位。
- type：属性类型。每种类型用一个UUID编号。
- value：属性值。
- permission：属性权限，分为无、可读、可写、可读写。

服务器储存了多个属性。当客户端向服务器发起请求时，服务器会把自己的属性列表发给客户端。随后，客户端可以向服务器读取或写入

某一个属性值。用读写的方式，通信双方实现了双向通信。

以智能手表为例。智能手表和手机配对后，手机可以用读的方式获得智能手表中某个属性下保存的步数，也可以用写的方式写入另一个属性负责的时间。在读写操作中，都是由客户端主动，服务器只能被动应答。ATT还提供了**通知**（Notification）的工作方式。当服务器改变了某个属性值时，可以主动通知订阅了该属性值的客户端。智能手表中的手势识别，就可以通过通知的方式告知手机。这样手机就可以实时地获知手势改变信息了。

GATT协议构建在ATT协议之上，为属性提供了组织形式。GATT协议的最小组织单元是**特征**（Characteristic），可以由数条属性组成。表12-1就是一个特征，用于传输红外测温获得的数据。这个例子来自一款可以进行蓝牙连接的硬件设备^[1]，该设备用BLE发送温度等传感器的测量数据。

表12-1 红外测温特征

句柄 handle	类型 type	值 value	权限 permission
0x24	0x2803	12:25:00:00:00:00:00:00 :00:00:B0:00:40:51:04:01 :AA:00:F0	R
0x25	0xAA01	00:00:00:00	RN
0x26	0x2902	00:00	RW
0x27	0x2901	54:65:6D:70:2E:20:44:61:74:61	R

特征的第一条是声明，其类型是0x2803。这条声明的value部分又可以细分为三部分。

- 最开始的0x12，称为**特征属性**（Characteristic Properties），是GATT协议层面上的权限控制^[2]。

- 随后的0x25，表示了特征数据所在的句柄。因此，0x25的属性值，就是红外温度的真正数值。我们顺着查看0x25的值，可以看到此时的读数为0。

- 剩下的部分包含了该特征的UUID，总共128位。写成UUID的顺序，即为 F000-AA01-0451-4000-B000-000000000000。除了128位的UUID，蓝牙官方还提供了16位的UUID可供使用。

可以看到，一个特征至少需要两个属性，一个用于声明，另一个用于储存它的数据。除此之外，特征还有被称为**描述符**（Descriptor）的额外描述信息。每个描述符占据一行。比如0x0027这个描述符，其属性值是：

54:65:6D:70:7E:20:44:61:74:61

翻译成ASCII就是：

Temp~ Data

Temp Data是**温度数据**（Temperature Data）的简写，所以这里说明了数据是温度数据。

此外，温度单位、测量频率等描述信息也经常会以描述符的形式放入特征中。在下一个特征声明出现前的属性，都是该特征的描述符。

再来看更高级的组织单位——**服务**（Service）。一个服务也有行属性作为声明，其类型UUID是0x2800。声明属性的值就是该服务的128位UUID。蓝牙官方也提供了16位的UUID，预留给特定的服务^[3]。在下一个服务声明出现前的属性都属于该服务，比如表12-2中从0x0023到0x002D的属性。

表12-2 0x0023到0x002D的属性

句柄 handle	类型 type	值 value	说明
0x23	0x2800	F000AA00 -0451-4000 -B000-000000000000	服务： 红外感温
0x24	0x2803	12:25:00:00:00:00:00:00 :00:00:B0:00:40:51:04:01 :AA:00:F0	特征： 红外感温 数据
0x25	0xAA01	00:00:00:00	
0x26	0x2902	00:00	
0x27	0x2901	54:65:6D:70:2E:20:44:61:74:61	
0x28	0x2803	...	
0x29	0xAA02	...	特征： 红外感温 设置
0x2A	0x2901	...	
0x2B	0x2803	...	
0x2C	0xAA03	...	特征： 红外感温 周期
0x2D	0x2901	...	
0x2E	0x2800	F000AA10 -0451-4000 -B000-000000000000	服务： 加速度
.....			

表12-2中包含了一个与红外温度计相关的服务。该服务包括了三个特征。第一个特征从0x24开始，到0x27结束。这个特征就是前面已经介绍过的传输红外感温数据的特征。第二个特征从0x28到0x2A，用于设置红外温度计参数。第三个特征是从0x2B到0x2D，用于设置测温频率。句柄0x002E之后，开始了一个新的服务。

服务和特征都是属性的组织形式。客户端可以向服务器请求服务和特征列表，然后对其进行操作。GATT还提供了**规范**（Profile）。一个规范可以包括多个服务。不过，规范并不像前面两者那样存在于服务器中。规范是一种标准，用于说明一个特型设备应该有哪些服务。比如，HID（Human Interface Device）这种规范，就说明了蓝牙输入设备应该提供的服务。

12.3 Bluez

我们用树莓派来深入实践前面学到的蓝牙知识。首先要在树莓派上安装必要的工具。BlueZ是Linux官方的蓝牙协议栈，你可以通过BlueZ提供的接口进行丰富的蓝牙操作。

Raspbian中已经安装了BlueZ，笔者使用的BlueZ版本是5.43，你可以检查自己的BlueZ版本：

```
$bluetoothd -v
```

低版本的BlueZ对低功耗蓝牙的支持有限。如果使用的Bluez版本低于5.43，那么请升级BlueZ的版本。

你可以用下面的命令检查BlueZ的运行状态：

```
$systemctl status bluetooth
```

笔者返回结果是：

```
● bluetooth.service - Bluetooth service
   Loaded: loaded (/lib/systemd/system/bluetooth.service; enabled)
   Active: active (running) since Sun 2017-04-23 19:03:08 CST; 1 day 6h ago
     Docs: man:bluetoothd(8)
 Main PID: 709 (bluetoothd)
    Status: "Running"
   CGroup: /system.slice/bluetooth.service
           └─709 /usr/lib/bluetooth/bluetoothd -C
```

可以看到，蓝牙服务已经打开，并在正常运行。

你可以用下面的命令手动启动或关闭蓝牙服务：

```
$sudo systemctl start bluetooth
$sudo systemctl stop bluetooth
```

此外，还可以让蓝牙服务随系统启动：

```
$sudo systemctl enable bluetooth
```

12.4 了解树莓派上的蓝牙

在Raspbian中，基本的蓝牙操作可以通过BlueZ中的bluetoothctl进行。该命令运行后，将进入一个新的Shell。这个Shell是由BlueZ提供的，与Linux系统的Shell不同。这个Shell中支持蓝牙相关的一些命令，比如输入：

```
list
```

将显示树莓派上可用的蓝牙模块，如：

```
Controller B8:27:EB:72:47:5E raspberrypi [default]
```

运行scan命令，开启扫描：

```
scan on
```

扫描启动后，用devices命令可以打印扫描到蓝牙设备的MAC地址和名称，如：

```
Device 00:9E:C8:62:AF:55 MiBOX3  
Device 4D:CE:7A:1D:B8:6A vamei
```

此外，还可以用help命令获得帮助，使用结束后，可以用exit命令退出bluetoothctl。

除了bluetoothctl，在系统Shell中可以通过hciconfig来控制蓝牙模块。比如，我们可以启动蓝牙模块：

```
$sudo hciconfig hci0 up
```

下面的命令可以关闭蓝牙模块：

```
$sudo hciconfig hci0 down
```

命令中的“hci0”指的是0号HCI设备，即树莓派的蓝牙适配器。

还可以用下面的命令来查看蓝牙设备的工作日志：

```
$hcidump
```

BlueZ本身还提供了连接和读写工具，但不同版本的BlueZ相关功能的差异比较大，而且使用起来不太方便，所以下面使用Node.js的工具来实现更深入的开发。

12.5 树莓派作为BLE外设

尝试用树莓派进行BLE通信。我们先把一个树莓派改造成BLE外设，同时它也将充当连接建立后的服务器。这个过程较为复杂，你可以借用Node.js下的bleno库。

首先，安装Node.js：

```
$curl -sL https://deb.nodesource.com/setup_5.x | sudo bash -
$sudo apt-get install nodejs
```

然后，安装bleno：

```
$mkdir ble-test-peripheral
$cd ble-test-peripheral
$npm install bleno
```

运行bleno中pizza的例子：

```
$sudo node node_modules/bleno/examples/pizza/peripheral
```

你可以在node_modules/bleno/examples/pizza/中看到源代码，或者到Github网站查看。

这个名为pizza的例子提供了一个关于披萨的服务，它的UUID是1333-3333-3333-3333-3333-333333333337。服务中包含了三个特征，分别是用于披萨饼选项、配料参数和烤披萨，如表12-3所示。

表12-3 特征

功能	权限	UUID
披萨饼选项	读/写	133333333333333333333333330001
配料参数	读/写	133333333333333333333333330002
烤披萨	写/通知	133333333333333333333333330003

通过这些特征，我们可以对树莓派进行BLE读写。读写操作会作用于一个代表披萨的对象。披萨饼选项如表12-4所示。

表12-4 披萨饼选项

数值	描述
0x00	正常
0x01	厚
0x02	薄

配料是一个8位的参数，如表12-5所示，每一位代表了一种配料。当这一位是1时，那么说明添加该配料：

表12-5 披萨饼配料

第 n 位	7	6	5	4
描述	SAUSAGE	BELL_PEPPERS	PINEAPPLE	CANADIAN_BACON
第 n 位	3	2	1	0
描述	BLACK_OLIVES	EXTRA_CHEESE	MUSHROOMS	PEPPERONI

因此，0x1A 代表了添加 MUSHROOMS、BLACK_OLIVES、CANADIAN_BACON，即蘑菇、黑橄榄、加拿大培根肉，味道应该不错。

对于烤披萨来说，写操作设定了烘烤的温度和时间。时间到了之后，中心设备会发出通知，告诉客户端烘烤完成。下一步将用另一个树莓派作为BLE中心设备。即使你没有另一个树莓派，你也可以用手机App^[4]来测试BLE外设。

12.6 树莓派作为BLE中心设备

我们拿另一个作为BLE的中心设备进行扫描，并发起连接请求。连接建立后，该服务器将充当客户端。和bleno对应，Node.js下有一个叫noble的项目，可以便捷地完成这一任务。首先，安装noble：

```
$mkdir ble-test-central
$cd ble-test-central
$npm install noble
```

noble中有一个同样名为pizza的例子，不过这个例子实现的是客户端。运行该例子：

```
$sudo node node_modules/noble/examples/pizza/peripheral
```

这个例子将自动执行扫描、连接、服务发现、数据传输的全过程。如果把bleno和noble部署到两个树莓派上，就可以在这两个树莓派之间进行蓝牙通信了。如果想自定义开发，那么可以在 `node_modules/noble/examples/pizza/` 上参考源代码，或者到Github查看。

12.7 树莓派作为Beacon

苹果在BLE的基础上推出了iBeacon协议。iBeacon使用了BLE的广播部分，但不建立连接。一个遵守iBeacon协议的外设被称为Beacon。Beacon会广播自己的身份信息和发射信号的强度。中心设备接到广播之后，除了可以获知Beacon的身份之外，还能通过信号的衰减算出自己与Beacon的距离。在一个典型的超市应用场景中，每件商品可以带上一个Beacon。消费者可以用手机看到自己周围有哪些商品，工作人员也可以用手机来清点货物。商家还可以在服务器上提供商品相关的质保、促销等信息。用户可以根据Beacon的编号，获得这些附加信息。

我们把配备了蓝牙模块的树莓派改造成一个Beacon。既然Beacon只使用了蓝牙中的广播，那么应该关闭树莓派的扫描，打开广播，并且不接受蓝牙连接。用下面的命令来关闭扫描：

```
$sudo hciconfig hci0 noscan
```

然后让蓝牙模块开始广播，并且在广播中不接受连接：

```
$sudo hciconfig hci0 leadv 3
```

把广播信息改为符合iBeacon协议的内容：

```
$sudo hcitool -i hci0 cmd 0x08 0x0008 1E 02 01 1A 1A FF 4C 00 02 15 63 6F 3F 8F  
64 91 4B EE 95 F7 D8 CC 64 A8 63 B5 00 01 00 02 C5
```

上面的命令附加了一串16进制信息。其中0x08说明了整条信息是蓝牙命令，0x0008说明后面的内容将作为广播信息。

1E是广播信息开始的标志。按照蓝牙通信的规定，广播信息最多有31个字节。1E后面的广播信息分为两组。

- 第一组：0201 1A
- 第二组：1A FF 4C 00 02 15 63 6F 3F 8F 64 91 4B EE 95 F7 D8 CC 64 A8 63 B5 4B EE 00 02 C5

每一组开始的一个字节说明了该组信息的长度。02说明了两字节，1A说明是26个字节。随后一个字节说明了该组信息的类型。第一组的01说明了该组信息是蓝牙控制标志，第二组的FF说明了该组是蓝牙制造商相关信息。

我们来看第二组信息的细节：

- 4C 00是制造商信息，即苹果。
- 02 15是iBeacon协议标识。
- 63 6F 3F 8F 64 91 4B EE 95 F7 D8 CC 64 A8 63 B5是设备的UUID，通常是用户编号。
- 00 01是主编号（Major）。
- 00 02是次编号（Minor）。

把UUID、主编号、次编号合在一起，我们可以确定Beacon的唯一身份。

最后的C5说明了蓝牙信号强度，即在1米处测得的该Beacon的RSSI值。中心设备把接收到的信号强度和该信号强度对比，就可以知道信号衰减了多少，从而推算出自己与Beacon的距离。由于我这里写入的C5没有经过校准，所以距离测量可能不准确。

用手机上探测Beacon的App来测试。当进入树莓派的广播范围时，应用就会显示出手机距离树莓派的距离。

使用结束后，可以用下面的命令停止广播：

```
$sudo hciconfig hci0 noleadv
```

用下面的命令来恢复扫描：

```
$sudo hciconfig hci0 piscan
```

[1] Texas Instruments公司的SensorTag。

[2] 可参考https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.attribute.gatt.characteristic_declaration.xml。

[3] 可参考<https://www.bluetooth.com/specifications/gatt/characteristics>。

第13章 你是我的眼

树莓派官方出品有小型摄像头，用于录制视频或拍摄图片。树莓派加上小型摄像头，就构成了一个好玩的移动摄影装置。

最新的官方摄像头版本是V2，配有8M像素的Sony IMX219感光板，可以满足一般的摄影摄像需求。V2摄像头又可以分为两款。一款摄像头用于正常的可见光拍摄，名为Pi Camera V2；另一款摄像头带有红外夜视功能，名为Pi NoIR Camera V2。本章的内容同时适用于这两种摄像头。

13.1 摄像头的安装与设置

树莓派摄像头安装在一个方形的电路板上，从电路板上伸出一根柔软的排线。我们需要把摄像头的排线插入树莓派上的“camera”接口。首先抬起接口的盖子，然后放入排线，最后把盖子重新装回去，如图13-1所示。

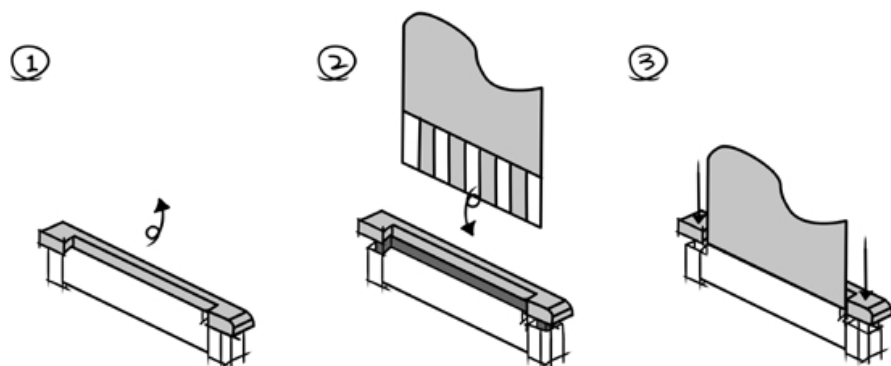


图13-1 摄像头的安装过程

安装好摄像头后，打开树莓派。首先更新Raspbian系统的软件源并升级：

```
$sudo apt-get update && sudo apt-get upgrade
```

其次，我们要在树莓派设置中，启动摄像头模块，用命令进入设置页面：

```
$sudo raspi-config
```

在设置页面中选择启动摄像头。

13.2 摄像头的基本使用

设置完成后，摄像头就可以工作了。Raspbian提供了raspistill和raspivid两个工具，分别用于获得图片和视频。

1.用摄像头拍照

我们用raspistill拍照：

```
$raspistill -o image.jpg
```

拍照获得的图片保存为文件 **image.jpg**。你可以用文件管理器找到并查看该照片。

2.用摄像头录视频

我们可以用raspivid录视频：

```
$raspivid -o video.h264 -t 10000
```

获得10秒H.264压缩格式的视频，存入文件 **video.h264**。

我们可以把H.264文件转换为更常见的MP4视频文件格式。GPAC是一款多媒体框架，提供了视频格式转换的功能。安装GPAC：

```
$sudo apt-get install gpac
```

用GPAC中的MP4Box把文件转换为 **video.mp4**：

```
$MP4Box -fps 30 -add video.h264 video.mp4
```


大部分视频播放器都可以播放MP4视频。在Raspbian中播放 **video.mp4**：

```
$omxplayer video.mp4
```

这里的omxplayer是Raspbian中的视频播放器。

13.3 用VLC做网络摄像头

除了直接录制视频文件，树莓派的摄像头还能拍摄流媒体，用于网络播放。Raspbian下有很多工具可以实现这一功能，这里介绍VLC的用法。

VLC是大名鼎鼎的视频播放软件，支持包括Raspbian在内的多个平台。在Raspbian下安装VLC，作为流媒体的服务器：

```
$sudo apt-get install vlc
```

利用Linux下的管道机制，把raspivid拍摄的内容导入VLC：

```
$raspivid -o - -t 0 -n -w 480 -h 480 | cvlc -vvv stream:///dev/stdin --sout '#standard{access=http,mux=ts,dst=:8160}' :demux=h264
```

选项-n说明了不显示预览窗口。随后VLC作为服务器，将流媒体送到树莓派的8160端口。同一网络下的任意装有VLC的设备都可以通过访问树莓派的IP地址和8160端口来播放摄像头拍摄的内容。比如树莓派在笔者的局域网中的IP地址是192.168.1.27，那么在手机版VLC的网络媒体源中输入下面网络源：

```
http://192.168.1.27:8160
```

就可以在同一局域网下查看该网络摄像头的实时视频。

我们用树莓派制作了一个可移动的网络摄像头。更进一步，我们可以通过隧道的方式把视频内容绑定到某个互联网服务器上，从而在互联网范围内发布该网络摄像头。实现隧道的方法已经在第8章中介绍过了。

13.4 用Motion做动作捕捉

Motion是Linux下一款轻量级的监控软件。在日常工作模式下，Motion可以提供网络摄像头的功能。在拍摄过程中，当画面发生变动时，Motion可以保存动作发生时的图片和视频。动作捕捉的功能对于安保监控有很大帮助。我们配合Motion来使用树莓派摄像头。

1.使用Motion

首先，下载安装Motion：

```
$sudo apt-get install motion
```

修改 */etc/default/motion* ，更改设置，让Motion启动后台的守护进程：

```
start_motion_daemon=yes
```

然后，修改Motion的配置文件 */etc/motion/motion.conf* ，更改下面几个值为：

```
daemon on
```

让Motion作为背景的守护进程运行。

```
stream_localhost off
```

如果是on，那么只有树莓派自己可以看到流媒体。如果是off，那么网络上的其他主机也可以看到。

```
stream_maxrate 30
```

表示流媒体的帧速率最大为每秒30帧。

```
framerate 30
```

表示摄像头捕捉视频的帧速率为每秒30帧。

选项修改好之后，就可以启动Motion了：

```
$sudo service motion start
```

现在摄像头已经在录制流媒体了。在同一局域网下，用浏览器打开192.168.8.113:8081这一网址，就可以直接看到即时拍摄的流媒体。此外，Motion还可以进行动作捕捉。如果你在摄像头前挥手，那么Motion会捕捉这一动作，并把相关的图片和视频存储在目录 ***/var/lib/motion*** 之下。

2.Motion的其他设置

Motion的主要设置都在 ***/etc/motion/motion.conf*** 文件中。除上面我们修改的配置外，文件中还有许多其他选项，这里选择一些重要的配置进行介绍。

(1) **target_dir**：该选项的默认值为 ***/var/lib/motion*** 。这就是Motion存储动作捕捉结果的地方。Motion的进程是以用户motion的身份运行的，所以用户motion必须对该目标文件夹有写入权限。本书的第18章会介绍用户权限的相关内容。

(2) **stream_port**：流媒体的输出端口，默认值是8081，也就是我们刚才访问流媒体的端口。如果有需要，可以更改输出端口。

(3) **threshold**：动作捕捉阈值，默认值为1500。如果有超过阈值的像素点发生变化，那么认为有动作发生。

(4) **videodevice**：该项默认为路径 ***/dev/video0*** 。这个路径对应了默认的视频设备。如果你无法在 ***/dev*** 下找到 ***video0*** ，那么可以尝试加载V4L2驱动来解决问题：

```
$sudo rpi-update  
$sudo modprobe bcm2835-v4l2
```

第3部分 进入Linux

作为一款经典的开源操作系统，Linux在移动设备、网站服务器、超级电脑上都很常见。树莓派上的Raspbian系统其实也是一款Linux系统。因此，树莓派正是学习Linux相关知识的好工具。对于初学者来说，Linux等价于一大堆命令。为了避免这种情况，本书的介绍偏重于功能模块的设计理念，而不是命令参数之类的细节。

第14章 Linux的真身

我们经常用“Linux”来指代整个Linux操作系统。但对于不同的人来说，“Linux”指代的含义又有所区别。说到托瓦兹写了Linux系统，意思是说他写了Linux的内核。而说到安装Linux系统，大多数时候是指安装了Linux的一个厂商版本。首先来区分描述Linux的几个关键词：内核、GNU和厂商版本。

14.1 什么是内核

Linux系统有狭义和广义两种定义。狭义来说，Linux实际上指Linux**内核**（kernel）。广义来说，Linux是指以内核为基础的，包括了各种应用软件在内的Linux**发行版**（Distribution）。如果不加区分地说Linux系统，就很容易造成混淆。

Linux系统可以简单地区分为内核程序和应用程序两个部分。内核程序在Linux启动后就一直运行着。这个程序有权调配所有的计算机资源：运算资源、存储资源、接口资源等。内核会根据应用程序的需求，提供实现应用程序所需的资源。从这个角度看，内核就好像服侍应用程序的“大内总管”。当然，内核也不是一味迎合，它还有一套调配资源的规则。如果应用程序提出无理需求，那么内核也会毫不犹豫地拒绝。托瓦兹编写的Linux系统，实际上只有Linux内核。他所开源的，也正是Linux内核的代码。

内核程序之外的就是应用程序。应用程序只有在内核启动后才会运行。大多数的应用程序必须经用户调用才可以启动。当然，用户不一定要手动调用。就拿开机时来说，内核启动后会运行一个初始化脚本，调用常用的应用程序，比如bash或图形化桌面。每个应用程序都能实现某用户需要的功能，比如作为网络浏览器的Firefox、作为邮件客户端的Thunderbird、作为多媒体播放器的VLC。

一个运行中的Linux系统，往往同时运行着多个应用程序。内核管理着这些应用程序。内核会给每个应用程序独立的内存空间和运算时间，从而让应用程序可以同时运行。不同的应用程序有不同的权限，以便调用不同级别的内核功能。当多个应用程序调用同一个硬件设备，如打印机时，内核必须决定其优先级，以免出现多个应用程序同时打印在一张纸上的混乱情况。无论如何，没有任何应用程序可以像内核一样全面掌控计算机资源。

内核程序和应用程序的区分并非Linux独有的，大多数现代的系统都会有此结构。当然，我们也可以制作一个操作系统，允许应用程序直接调用计算机资源。这样还可以省去运行内核程序的开销，应用程序甚至可以达到更高的运行效率。很多功能简单的嵌入式系统，如智能手环等硬件设备，就是这么做的。但在一个多用户多应用程序的复杂系统中，内核的缺失会带来很多问题。一个应用程序对计算机资源的调用很可能影响到其他的程序。缺了内核的中心调度，程序之间会相互干扰，整个系统混乱不堪。内核与应用程序的关系，如图14-1所示。



图14-1 内核与应用程序的关系

从软件开发的角度看，如果每个应用程序都要直接操纵底层硬件，那么编写应用程序的程序员就必须熟知硬件知识，这将大大增加程序开发的难度。要知道，即使是鼠标这样简单的外设，其编程也需要多位高级程序员的通力合作。而像CPU、内存这样复杂的硬件，相关文档的工作量更是惊人。光是一个处理器的规格书，就有数百页。内核以上千万行代码为代价，提供了一套接口。应用程序的开发人员只要熟知这一套接口，就能轻松地开始程序开发。以Linux为例，它提

供的接口可以总结为300多个函数接口，其中常用的只有几十个。只要掌握了这一套接口，应用程序的程序员就足以发挥内核那千万行代码才能实现的功能。

此外，Linux的接口是按照POSIX（Portable Operating System Interface）标准制作的。由于其他UNIX系统同样遵从POSIX标准，所以Linux可以很容易地和其他UNIX系统互通。为Linux系统编写的应用程序，只要简单修改，就可以应用到其他的UNIX系统，比如Solaris、FreeBSD和基于FreeBSD的苹果公司的Mac OS。这样的通用性受益于内核程序和应用程序的分离。因此，内核不但可以合理调配计算机资源，还简化了应用程序的开发。

14.2 什么是GNU软件

Linux程序的最初流行，与一套名为GNU的应用软件密不可分。如果说Linux是开源运动的明星，那么GNU算得上是开源运动的鼻祖。早在1983年，GNU项目就已经诞生。GNU是“GNU's Not UNIX”的缩写。这个名称是对传统商用UNIX系统的宣战。GNU项目旨在创造一套自由免费的UNIX系统。GNU标志如图14-2所示。



图14-2 GNU标志

按照创始人理查德·斯托曼（Richard Stallman）的计划，GNU系统应该包括内核和应用程序。当托瓦兹写出Linux内核时，GNU已经孵化了很多好用的开源应用程序，并且已经在多个UNIX平台上得到广泛使用。这些应用程序包括了C语言编译器gcc、作为Shell的bash、文本编辑器nano等。因为这些应用程序都是按照UNIX接口编写的，所以很容易移植到Linux系统上。因此，托瓦兹在发布Linux内核时，也在Linux环境下编译了GNU的应用软件，来提高Linux系统的可用性。在绝大多数Linux系统上，GNU软件都成了一个必不可少的组成部分。另一方面，Linux内核的迅速流行，也让GNU放弃了自己的内核开发计划。

不过，尽管Linux内核和GNU关系密切，但两者并没有真正合为一体。因为主导内核开发的Linux基金会，和主导GNU开发的自由软件基金会，是两个独立的组织。Linux内核和GNU对开源软件的态度，也有不小的差异。不少GNU阵营的程序员认为，GNU软件对Linux贡献巨大，因此Linux应改名为GNU/Linux。但托瓦兹认为，内核程序和GNU应用程序是两个不同层面上的独立产物，没有必要混为一谈。但这种闹哄哄的吵嚷并不影响Linux内核和GNU程序在用户那里实质性的共存。这也正是开源运动的魅力所在。尽管整个开源运动分裂为数不清的软件项目，但用户总可以根据自己的需要来组合使用。

14.3 Linux的发行版

即使有了内核和GNU软件，Linux的安装和编译并不是简单的工作。此外，对于商用的Linux来说，后期维护也让人头疼。所谓的厂商就是一些Linux服务商。他们提供Linux运行所需的额外服务，从而让客户可以更容易地使用Linux系统。Linux操作系统在很多专业领域应用广泛，这些厂商基于其提供的服务可以赚取丰厚的利润。

Linux厂商一般都提供咨询和维护服务。咨询服务可以帮你分析Linux是否适合你的业务和应用，以及如何更好地在你的工作流程中使用Linux。维护服务则包括了安装、故障排查、升级等，从而让Linux系统可以长期稳定运行。为了便于服务，这些厂商会在Linux内核和

GNU的基础上，开发自己的软件并调整配置，以便更好地进行客户支持。最终，厂商会把软件和配置整合在一起，形成发行版。大部分用户使用的都是厂商提供的发行版。这些发行版极大地提高了系统的易用性。

Linux服务市场有不少大玩家。红帽早已是上市公司。IBM是Linux设备最大的供应商，同时它的咨询业务很大一部分也来源于提供Linux相关的支持。我们所熟知的Android操作系统是Google提供的一个发行版。树莓派的Raspbian，也是由树莓派官方提供的一个发行版。

这里主要介绍在PC上比较流行的Linux发行版。首先是三大家族。

1.红帽家族

红帽公司自20世纪90年代创立以来一直是最重要的Linux厂商之一。1999年，红帽公司上市，成为Linux的著名商业案例。直到今天，红帽依然是Linux厂商中规模最大的一家。

- Red Hat Linux:大名鼎鼎的红帽Linux，现在已经完结，其后的几个Linux版本都以此为基础。

- Red Hat Enterprise：企业级的红帽Linux，主要面向服务器。作为商业版，它有比较好的配套软件和技术支持。它的教材也堪称经典。

- Fedora：由社区维护，去除了一些商业软件。红帽实际上赞助了这个项目，以便以此作为技术测试平台。

- CentOS：这个版本虽然不来自红帽公司，但它由红帽公司公开的源码组成。CentOS是免费版本，由社区维护，和红帽完全兼容。CentOS版本升级较慢，所以适合不愿意频繁升级的情况。因此，CentOS在多用户服务器上应用较广。

2.SUSE家族

SUSE由德国公司SUSE Linux推出。由于最初服务于德国市场，所以SUSE在欧洲比较流行。SUSE系列比较有特色的是YaST2软件。

YaST2有图形化界面，主要用于设置和管理SUSE系统，对初级的Linux用户来说比较方便。

- SUSE Linux Enterprise：商业版本，和红帽商业版类似。
- openSUSE：SUSE的免费版本。以前SUSE不是很重视这个免费版本，支持不好。现在SUSE官方对该版本的态度大大转变，支持力度增加了很多。但就笔者个人的使用体验来说，还是觉得社区支持不足。

3.Debian家族

Debian是最早的Linux发行版本之一。这个家族的Linux版本都以社区维护为基础，具有非盈利的倾向。其中的Ubuntu等已经开始了一些商业尝试，但并没有因此影响到免费用户的体验。

- Debian：完全免费，社区维护的Linux版本，有很大的用户群，所以遇到问题，基本都可以找到社区用户的支持。
- Ubuntu：由一个基金提供支持的免费Linux版本。它继承自Debian，界面友好。对于初次在PC上安装Linux的用户来说，这是最适于安装的版本。
- Mint：基于Ubuntu。它提供了更加丰富的预装应用，以减少用户搜索并安装应用的麻烦。其使用的应用版本比较新，可能不是很稳定。
- Raspbian：和Ubuntu一样，Raspbian继承自Debian。它是由树莓派官方推出的发行版，对树莓派有很好的支持。

除了上面提到的三大家族外，Linux还有如下版本。

- Gentoo：基于源码的版本，给用户很大的自由度。为用户提供大量应用程序的源码，可以在用户的系统上重新编译建造，需要一定的系统配置知识。
- ArchLinux：推崇简洁，避免不必要和复杂的修改，是一个轻便灵活的版本，其配置文件有良好的注释。
- Mandriva：一个很方便用户使用的版本，其目标是使新用户更容易使用Linux。

- Slackware：它的特点是稳定。它只包含稳定版本的应用程序，对于初级用户不是很友好。

- TurboLinux：在亚洲比较流行。它是商业版本，提供技术支持和咨询服务。

Linux发行版本数目众多，这里介绍的只是市面上常见的版本。如果想了解更多，可以在DistroWatch网上查询。该网站不但提供了各个发行版的介绍，还会发布它们的最新消息。

本章区分了Linux经常与混用的几个名词：内核、GNU和发行版本。尽管人们有时不加区分地把它们统称为Linux，但这三者的含义差别很大。了解了三者的区别，才能听明白别人说的是哪一个Linux。

第15章 你好，文件

对于计算机来说，所谓的数据就是0和1的序列。Linux上的文件提供了数据存储的基本单元。此外，文件还以目录的形式组织起来，以使用户能迅速找到所需数据。本章将深入了解文件的组织方式。

15.1 路径与文件

文件和文件组织构成了一个**文件系统**（File System）。Linux的文件系统是一个树状结构，整个文件系统有个共同的起点，就是树状结构的顶端，如图15-1所示。Linux把这个起点称为**根目录**（Root Directory），用符号/表示。

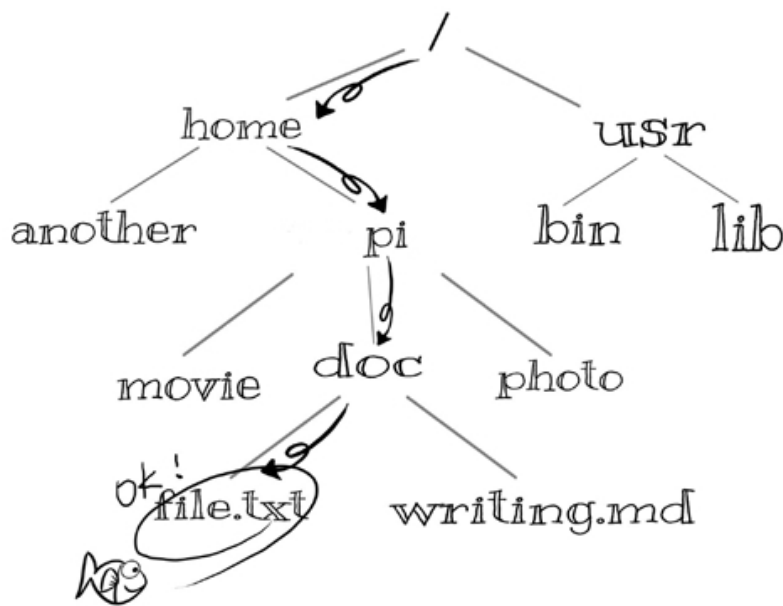


图15-1 树状结构的文件系统

文件树的末端可能是一个普通文件，用于存储数据，比如 *file.txt*。这个树上的节点还可能是一个目录，从而提供归属关系。通过归属关系，目录之间分为层级。目录 *pi* 称为 *home* 的**子目录**（Child

Directory），而目录 *home* 是 *pi* 的父目录（Parent Directory）。如果从该树中截取一部分，比如从目录 *pi* 开始往下，实际上也构成一个有单一起点的子文件树。

要找到一个文件，除了要知道该文件的文件名，还需要知道从根目录到该文件的所有目录名。从根目录开始的所有途径的目录名和文件名构成了一个路径。在图15-1的文件系统中，从顶端的根目录 */*，沿箭头标出的路径，经过目录 *home*、*pi*、*doc*，最终可以找到文件 *file.txt*。因此，为了找到文件 *file.txt*，我们需要知道完整的路径，也就是绝对路径 */home/pi/doc/file.txt*。

15.2 目录

在Linux系统中，目录把文件组织起来。其实，目录本身也是一种特殊的文件，而 */home/pi/doc* 是指向目录文件 *doc* 的绝对路径。我们可以使用 *file* 命令来获取文件类型：

```
$file /home/pi/doc
```

该命令将返回：

```
$/home/pi/doc: directory
```

也就是说，*/home/pi/doc* 是一个目录文件。

作为对比，我们用 *file* 获得 */home/pi/doc/file.txt* 的类型：

```
$file /home/pi/doc/file.txt
```

将返回：

```
/home/pi/doc/file.txt: ASCII text
```

/home/pi/doc/file.txt 是一个ASCII编码的文本文件。

目录文件中的内容以条目的形式存在。它至少包含以下条目：

```
.      指向当前目录
..     指向父目录
```

除此之外，目录文件中还有属于该目录文件的文件名，比如 **/home/pi/doc** 中有如下内容：

```
../
./
file.txt
writing.md
```

Linux理解一个绝对路径的方式如下：先找到根目录文件，从该目录文件中读取 **home** 目录文件的位置，然后从 **home** 文件中读取 **pi** 的位置。通过一层层目录的查询，Linux最终会找到目录 **doc** 中 **file.txt** 的位置。因为目录文件中都有当前目录和父目录的条目，所以我们可以 在绝对路径中加入“.”或者“..”来表示当前目录或者父目录，比如 **/home/./pi/doc/..** 与 **/home/pi** 等同。

此外，当Linux程序运行时，会维护一个名为**工作目录**（Present Working Directory）的变量。在Shell中，用pwd命令确定的当前目录，实际上就是Shell程序的工作目录。而所谓的变换目录，就是把一个新的目录存入该变量中，例如：

```
$cd /home/pi
```

有了工作目录，我们就可以用相对工作路径来创建绝对路径。例如，如果工作目录是 **/home/pi**，那么就可以用 **doc/file.txt** 来指示 **/home/pi/doc/file.txt**。在Shell中输入命令时，就可以用相对路径来替换绝对路径：

```
$ls doc/file.txt
```

由于相对路径借用了工作目录的信息，所以在大部分情况下，相对路径都比绝对路径简短。

15.3 硬链接

当目录文件中增加一个文件的条目时，就建立一个指向文件的**硬链接**（Hard Link）。一旦有了对应于文件的硬链接，这个文件就纳入了文件系统中。一个文件允许出现在多个目录中，这样，它就有多个硬链接。文件拥有的硬链接数目，称为文件在整个系统的**链接数**（Link Count）。当文件的链接数降为0时，说明文件已经孤立于文件系统之外。这样的文件会被Linux删除。

大多数情况下，一个文件只存在于一个目录之下，所以连接数为1。在这种情况下，一旦删除目录中该文件的条目，也就是删除一个硬链接，那么该文件就会被删除。如果是图15-1中的文件结构，那么使用删除硬链接的unlink命令：

```
$unlink file.txt
```

file.txt 的条目将从目录文件 */home/pi/doc* 中删除，文件的链接数降为0。在这种情况下，unlink效果等同于删除文件。

如果给同一个文件在新的目录下建立硬链接，那么该文件的链接数就超过了1。我们用ln命令来创建硬链接：

```
$ln file.txt /home/pi/movie/another_file.txt
```

原来的文件在 */home/pi/doc* 目录下。通过新建硬链接，*/home/pi/movie* 也会多出一个硬链接记录。该记录的文件名是 *another_file.txt*，但实际上它和 *file.txt* 是同一个文件。所以，对其中任意文件的修改，也会出现在另一个文件中。你可以用 *nano* 编辑修改 *file.txt*。注意，修改会直接出现在 *another_file.txt* 中。

此时，再使用unlink命令：

```
$unlink another_file.txt
```

还可以通过 */home/pi/doc/file.txt* 找到该文件，文件并没有被删除。实际上，Linux中的rm命令和unlink命令功能相同，你可以试试

看。

15.4 软链接

同一文件的多个硬链接，会破坏树状的文件系统。因此，Linux系统并不鼓励手动创建硬链接。在必要的情况下，你可以用**软链接**（Soft Link）的方式，在多个目录下创建指向同一文件的链接。

软链接不影响文件的链接数。软链接本质上是一个文件，它的文件类型是symbolic link。在这个文件中，包含有链接指向的文件的绝对路径。当读写该文件时，Linux会根据软链接中的绝对路径把读写操作导向软链接所指向的文件。与Windows系统的“快捷方式”类似，Linux的软链接就是Linux的“快捷方式”。

我们在ln命令中加上-s选项，来创建软链接：

```
$ln -s file.txt /home/pi/file-link.txt
```

/home/pi/file-link.txt 是一个软链接文件。你可以用file命令获知其文件类型：

```
$file file-link.txt
```

结果为：

```
file-link.txt: symbolic link to `/home/pi/doc/test/file.txt'
```

此时，用 **nano** 编辑 **file-link.txt**，相关的读写操作也会反映在原文件 **file.txt** 中。和硬链接不同的是，软链接不影响文件的链接数，也不会破坏文件系统的树状结构。因此，软链接在Linux中使用广泛。以Linux下常用的网络服务器程序Apache为例，它会安装许多配置文件，每种配置文件针对一种情况。但Apache只会把特定目录下的配置文件作为其要加载的配置。这时可以通过建立软链接的方式，把目标配置文件链接到该目录。这样可以避免很多对原始配置文件的误操作。

软链接本身是一个文件，但很多时候它又会指代一个原始文件。这种双重身份有时会造成困惑。我们用 *nano* 来编辑软链接，那么该操作会跟随链接指引，作用于原文件。如果用 *rm* 来删除软链接，那么删除操作不会跟随软链接。所以，删除软链接后，原文件依然存在。一个命令是否跟随链接指引，是由该命令的程序决定的。不过在大多数情况下，对文件本身的操作，如读写数据和复制文件等，会跟随指引。而对涉及文件所属目录的操作，如删除、移动等，则不会跟随指引。

15.5 文件操作

对于一个文件，可以有很多种操作。例如，用 *touch* 命令新建一个空的普通文件：

```
$touch empty.txt
```

我们还可以创建一个新的目录：

```
$mkdir good
```

删除一个空目录：

```
$rmdir good
```

在第5章中，我们已经了解了复制的 *cp* 命令和删除的 *rm* 命令。这些命令除了作用于单个文件，还可以作用于从某个目录开始的整个子文件树。比如复制整个目录 */home/pi/doc*：

```
$cp -r doc doc-copy
```

这样，从 *doc* 开始的整个子文件树都将复制到 */home/pi/doc-copy*。

同样，我们可以删除某个子文件树：

```
$rm -r doc-copy
```

在了解了文件系统之后，我们还可以发现，很多文件操作并非作用于文件本身。前面已经提到，文件删除操作实际上作用于文件所属的目录文件。再比如，移动文件：

```
$mv file.txt /home/pi/file1.txt
```

也就是在目录文件 `/home/pi/doc` 中减少一个 `file.txt` 条目，而在 `/home/pi` 中增加一个 `file1.txt` 的条目。整个操作过程只涉及两个目录文件。

本质上，我们对于文件本身可以进行的操作，就是**读取**（Read）、**写入**（Write）和**运行**（Execute）。读取是从已经存在的文件中获得数据。写入是向新的文件或者旧的文件写入数据。除了读写，文件还可以作为一个程序运行。在Linux的文件系统中，如果某个用户想对某个文件执行某一种操作，那么该用户必须拥有对该文件进行这一操作的权限。Linux的文件权限与用户密切相关，笔者将在第18章讲解用户时继续深入。

15.6 文件搜索

Linux操作系统提供了一些用于文件搜索的命令，如find命令。**find**命令会递归地遍历文件系统，搜选出符合条件的文件。在执行find命令时，还可以说明想要对目标文件进行的操作。命令find会在找到文件后执行指定的操作。命令的基本用法：

```
find path ... [expression]
```

参数path是需要搜索的目标目录。如果有多个目标路径，则可以将多个路径依次列出。expression是一个可选的表达式，说明要对目标文件进行的操作。表达式由**主操作**（Primary）和**运算**（Operand）组成。

下面看一个例子，用find打印硬盘上所有文件后缀名为.c的文件。

```
$find / -name "*.c"
```

这个命令中的表达式含有一个主操作，即-name "*.c"。该主操作会筛选文件名满足*.c格式的文件。通配符*表示任意长度的字符串。因此，*.c表示所有以c为后缀的文件。注意，Linux系统上有些文件和目录需要一定的权限才能读取，用普通权限运行上面的命令时可能会遇到权限错误。

增加否条件，打印当前目录所有后缀名不是.c的文件。主操作可以不止一个。当有多个主操作时，find命令会依次实现它们的功能。我们可以用-not来为筛选性操作取反，比如打印当前目录所有后缀名不是.c的文件：

```
$find . -not -name "*.c"
```

再比如，输出当前目录所有后缀名为.c文件的详细信息：

```
$find . -name "*.c" -ls
```

命令find的用法相当繁杂，具体用例可以参考 *find* 的文档。相比之下，命令locate要比命令find精简很多，它也能根据文件名来寻找文件。例如：

```
$locate grep
```

查找名为 *grep* 的文件。

```
$locate -i l*t
```

在这个命令中，选项-i代表忽略大小写。这个命令代表查找以l开头，以t结尾的文件。

注意，locate命令的文件查找不是实时的，这一点和实时遍历文件树的find命令不同。文件系统的信息提前存于一个数据库，locate命令在这个数据库中查找文件。可以用下面的命令来更新文件系统信息的数据库。

```
$sudo updatedb
```

本章介绍了Linux下数据存储的关键单元——文件。文件以文件树的形式组织起来，而那些用于组成文件树的目录，其实也是一种特殊的文件。最后，介绍了常用的文件搜索命令。

第16章 从程序到进程

计算机不止是存储数据的仓库，它还可以进行多种多样的活动，比如收发电子邮件、播放电影、陪人们下棋。应用程序给计算机带来了丰富的动作。Linux系统在教育层面的活动都以“进程”为单位进行。本章我们将初探进程。

16.1 指令

计算机实际上可以做的事情非常简单。我们给CPU发出**指令**（Instruction），CPU就会执行这些基础动作。指令通常由一串二进制的序列构成。CPU会识别并执行这些指令。每一款CPU都有一套指令集，比如ARM CPU使用的精简指令集。

一条指令能做到的事情很少，比如计算寄存器中两个数的和，又如把内存中的数据移入寄存器。**寄存器**（Register）是CPU的临时存储空间。在树莓派中，即使是计算内存中两个数的和，也需要多条指令。

- 指令1：把内存1号地址的数值放入寄存器a。
- 指令2：把内存2号地址的数值放入寄存器b。
- 指令3：对寄存器a和寄存器b中的数值求和，放入寄存器a。
- 指令4：把寄存器a中的结果放入内存3号地址。

整个过程就像是厨师做番茄炒蛋。厨师要先从库房的两个货架上分别拿来番茄和蛋放在案板上，然后用锅把两种原料炒在一起。在计算机中，内存像是库房，寄存器像是案板，而CPU中的运算单元就是炒菜锅，如图16-1所示。



图16-1 货架、案板和炒菜锅：内存、寄存器和CPU

除了搬运数据和计算，CPU指令还可以控制计算机内部的其他硬件乃至外设。早期程序员必须背熟CPU的指令集，然后用指令写程序。那个时候的程序员必须把任务分解成一条条CPU可以直接理解的指令。这样的程序称为**机器程序**（Machine Code）。机器程序可以用**汇编语言**（Assembly Language）编写。比如加法程序可以用汇编语言写出来：

```
MOV AX, [20H]
MOV BX, [10H]
ADD AX, BX
MOV [20H], AX
```

AX和BX代表了寄存器的两个位置，而[20H]和[10H]是内存中的两个位置。汇编程序里的每一行都直接对应了一条CPU可以解读的指令。MOV表示移动数据，ADD表示相加。程序会把内存中的两条数据移入寄存器，计算两条数据的和，再把结果移回内存。注意，这段程序简化了很多内容。根据CPU的不同，相同功能的汇编程序也会不同。

无论如何，我们已经从上面的汇编程序看到一种编程方式。写汇编程序时，程序员说明硬件级别的动作，所以汇编程序运行起来很快。这就像是赛车手开手动挡的汽车，可以充分发挥出汽车的性能，驾驶得更流畅，也更加省油。但是，一旦所要实现的功能变复杂，那么汇编程序

需要的指令数会快速增加。此外，由于程序在指令使用上没有限制，也经常会造成很多错误。这也如手动挡汽车，一旦挂挡不当，就很容易熄火。

16.2 C程序

程序员一边痛苦地写着汇编程序，一边探索简化程序编写的方法。他们很快发现，汇编程序的语句之间会有某些固定的组合模式。比如，计算机经常会重复执行某些特定任务。就拿高斯求和来说，从1开始，加2，加3……一直加到100。在整个过程中，程序就是反复执行相同的加法操作。

那么与其手动重复相同的指令，不如用“循环”这样的语法来表示重复执行的任务，让计算机自己去完成重复。因此，程序员们发明了高级语言，用一些特殊的语法来抽象某些常见的指令组合。Linux系统的大部分程序，正是由C语言这一高级语言写成的^[1]。

先来看一个C语言的程序文件，这个文件的名字是 **demo.c**，你可以用nano来写C语言文件：

```
#include <stdio.h>

int main(void) {
    int a;
    int b;
    int c;

    a = 1;
    b = 3;
    c = a + b;          /*加法和赋值*/

    printf("sum is %d", c); /*调用函数 */
    return 0;
}
```

C程序用{}来表示一个程序块。上述程序定义了函数main。**函数** (Function) 是对一个程序块的抽象。函数main是C语言中的特殊函数。

当程序运行时，会自动调用其中的main函数。所谓的函数调用就是执行函数包含的程序块。函数运行完成后会有一个**返回值**（Return Value）。正如main前面的int表明的，main函数的返回值是**整数类型**（Integer）。

函数main的调用是自动进行的。除此之外，其他函数必须在某个语句中被调用，才可以执行。比如，函数main中调用了函数printf。函数printf会在屏幕上打印括号中的内容。当main函数执行到这一句时，printf函数就会被调用。写程序时，只要写清楚函数名，就可以调用一个功能复杂的程序块。

函数中创建了3个**变量**（Variable），分别用字母a、b、c表示。变量用于存储特定类型的数据。3个变量都是整数类型，用int表示，可以用来存储11、-24或1986这样的整数。变量是主存储器的一块空间，可用来存储数据。3个整型变量可以存储3个整数。

创建了变量之后，我们就可以往变量中读取或写入数据，例如：

```
c = a + b
```

就读取变量a和变量b中的数据，将它们相加，再存入变量c。“=”是**赋值**（Assignment）符号，用于把数据写入变量。

这段程序的功能，就是计算整数相加的结果，并以特定格式打印出来：

```
sum is 3
```

我们再来看高斯求和的实现方式：

```
int calculate_sum(int first_element(int first_element, int last_element, int
count) {
    int partial_sum = first_element, int last_element;
    int sum = partial_sum * count;
    return sum;
}
```

这一段C程序，和16.1节中的汇编语言的功能类似，但语法上区别显著。一方面，高级语言提供了很多便利。比如函数语法允许程序员来代表复杂的程序块，从而提高了程序的可复用性。另一方面，高级语言也

给出了很多限制，比如通过变量类型，给不同的变量以特定的内存空间，从而减少了出错的可能。通过提供便利和限制，高级语言提高了程序员的生产力。

16.3 程序编译

因为C语言中包含了很多抽象语法，所以计算机不能直接理解C语言的语法。高级C语言程序必须先编译成汇编程序，再转成机器程序运行。我们可以用gcc来编译一个C程序，比如：

```
$gcc demo.c
```

编译完成后，当前目录下会出现一个名为 **a.out** 的二进制可执行文件。用下面的方式执行该文件：

```
$/a.out
```

程序运行，在Shell中打印：

```
sum is 4
```

我们看到，计算机按照程序的描述执行了特定的活动，即计算两个数的和并打印出来。

C语言的编译是把程序员可读的C语言文本，翻译成计算机可读的机器程序。编译产生的 **a.out** 就是可执行文件，内含二进制文本。计算机可以直接读懂并执行该程序。这个二进制文本其实就是指令式的程序。通过apt-get下载的应用，大部分都是已经编译好的二进制可执行文件。我们可以直接使用这些程序，免去了编译的步骤。

但有些时候，软件商店中没有我们需要的软件，那么我们不得不从源代码出发，对程序进行编译。大部分Linux应用的编译都比上面例子的过程复杂。一般来说，源代码中还包含了编译大型工程所需的辅助文件。按照惯例，一般会有一个名为 **configure** 的脚本用于设置。编译的第一步，就是运行该脚本，根据提示进行设置：

```
$/configure
```

随后，你需要运行make命令：

```
$make
```

命令make会根据工程中的Makefile来解析代码文件之间的依赖关系^[2]。通常来说，一个工程会包含多个C语言程序。由于C语言可以跨文件地调用函数和变量，因此在编译时，代码文件之间相互依赖。命令make会根据依赖关系来编译文件。

最后，把编译好的二进制可执行文件放到 *configure* 设定的目标路径中：

```
$sudo make install
```

16.4 看一眼进程

虽然程序规定了活动的动作，但是应用程序并不等于**进程**（Process）。进程是程序的一个具体实现。我们只有运行程序，才能产生一个进程。程序和进程的关系，类似于食谱和做菜的关系。对于一个厨师来说，只有食谱没什么用，只有按食谱的指点一步步实行，才能做出菜肴。进程是执行程序的过程，类似于按照食谱真正去做菜的过程。

在Linux系统中，我们可以用ps命令来查询正在运行的进程：

```
$ps -eo pid,cmd
```

在这个命令中，-e选项表示列出全部进程，-eo pid, cmd选项表示我们需要的信息。执行结果的一部分示例如表16-1所示。

表16-1 执行结果

PID	CMD
1	/sbin/init splash
2	[kthreadd]
.....	
456	/usr/sbin/cron -f
457	/usr/sbin/rsyslogd -n
.....	
12509	ps -eo pid,cmd

在ps返回的结果中，每一行代表了一个进程。每一行又分为两列。第一列是一个整数，即**进程** ID（PID，Process IDentity）。每一个进程都有唯一的PID来代表自己的身份。无论是内核，还是其他进程，都可以根据PID来识别出该进程。第二列CMD是进程所对应的程序，以及运行时传递给程序的参数。

第二列中有一些由中括号括起来的。它们是内核的一部分功能，被打扮成进程的样子以方便操作系统管理。此外，PID为1的进程一定是由 */sbin/init* 程序运行而形成的。当Linux启动的时候，init是系统创建的第一个进程，这个进程会一直存在，直到关闭计算机。其他的进程就是常见的应用进程，例如cron进程和ps进程。

同一个程序可以执行多次，从而产生多个进程。即使一个程序的进程还没有完成，我们还是可以用同一程序运行出更多的进程。操作系统的一个重要功能就是管理进程，为进程提供必需的计算机资源，比如，为进程分配内存空间，管理进程的相关信息等。

[1] 本书不会深入讲解C语言的语法，但你可以参考附录C中的C语言语法摘要。

[2] Makefile的更多内容，可参考附录D Makefile基础。

第17章 万物皆是文本流

数据是计算机最宝贵的财产。在Linux中，**文本流**（Text Stream）是不同程序、不同文件之间的数据桥梁。通过这一数据桥梁，Linux的不同模块之间可以方便地进行协作。文本流是UNIX阵营的一大特征，也是UNIX系统备受称赞的一个设计。

17.1 文本流

在计算机中，所谓的数据就是0或1组成的二进制序列，每个0或1占一位。Linux系统对0和1的序列进行了分割，以**字节**（Byte）来作为数据单位。一个字节对应八位。比如下面一个八位的二进制序列就是一个字节：

01100001

八位的二进制数字会落在十进制从0到255的范围内。二进制的10011100转换成十进制数，就是97。

利用ASCII编码^[1]可以把这—个字节转换成为一个字符，即字母“a”。ASCII编码把从0到255的数字对应为英文字母、数字和常用符号。因此，一个字节总可以转换成一个ASCII字符。因为Linux以字节为单位分割数据，所以Linux中的数据完全可以用字符的形式表示出来，也就是所谓的文本。

当然，在计算机眼里，以位为单位或以字节为单位并没有多大差别。Linux用字节为单位，并不是为了机器。相对于以位为单位的二进制数据，以字节为单位的数据可以转换成**人类可读**（Human Readable）的字符。这样，无论是计算机配置信息，还是别人写的一首诗，用户都可以了解其含义。当然，并不是所有的数据都是设计来让人读懂的。比如，编译好的二进制文件是给机器读的。打开二进制文件，虽然也能看到一个一个字符，但这些字符并不能组成有意义的文本。但Linux系统依然以字节为单位处理这些二进制文件，不会特殊对待这些写给机器看的文

件。所有文件都是统一的形式，都能以相同的方法存储，也能共用一套处理工具，从而减少程序开发的难度。

存储文本的文件，就相当于一个个存储数据的房子。在Linux的设计哲学中，一向有“**万物皆是文件**（Everything is a file）”的说法。一般地存储用户数据的文件自不必说。表示文件位置的目录，也是保存在Micro SD卡上的一种文件。此外，系统的配置文件、软链接，也都是存储在存储设备中的文件。上述的文件不仅有存储数据的功能，而且可以读写数据。

在计算机系统中，除了存储设备，还有很多其他设备有读写数据的需求。第11章中的GPIO和UART端口都有读写功能。Linux把所有读写数据的对象都当作文件。在Linux中，我们通过操作设备文件，就可以和设备进行数据交流。在UART编程中，我们就通过 `/dev/AMA0` 这一文件和UART端口的设备直接对话。在 `/dev` 目录下，还可以找到很多其他的设备文件。

由于文件总和数据存储联系在一起，因此托瓦兹把“万物皆文件”的说法改为“**万物皆是文本流**（Everything is a stream of bytes）”。系统运行时，数据并不是在一个文件里定居。数据会在CPU的指挥下不断地流动，就好像一个勤劳的上班族。有时数据需要到办公室上班，因此被读入内存；有时会去酒店休假，因此传送到外部设备；有时数据需要搬个家，转移到另一个文件。在这样跑来跑去的过程中，数据像是有序流动的水流，我们叫它文本流。文本流有以下特性。

- 文本性：数据以字节为单位，可以转换成文本。
- 有序性：数据的前后顺序不会错乱。
- 完整性：数据内容不会丢失。

如果看过电影《骇客帝国》，那么一定会对屏幕上的文本流印象深刻。Linux用文本流的方式，为计算机不同模块之间的数据交换铺平了道路。文本流是不同模块之间进行数据交换的契约。每个应用程序都要确保自己发出的文本流有序且完整，其他应用程序接收到文本流时则不用为数据错乱头痛。文本流如图17-1所示。



图17-1 文本流

17.2 标准输入、标准输出、标准错误

文本流存在于Linux的每一个进程中。当Linux启动一个进程时，会自动打开三个流的端口：**标准输入**（Standard Input）、**标准输出**（Standard Output）和**标准错误**（Standard Error），这三个端口类似于入口、出口、紧急出口，如图17-2所示。进程经常会通过这三个端口进行输入和输出。当然，虽然一个进程总会打开这三个流，但进程会根据需要有选择地使用。

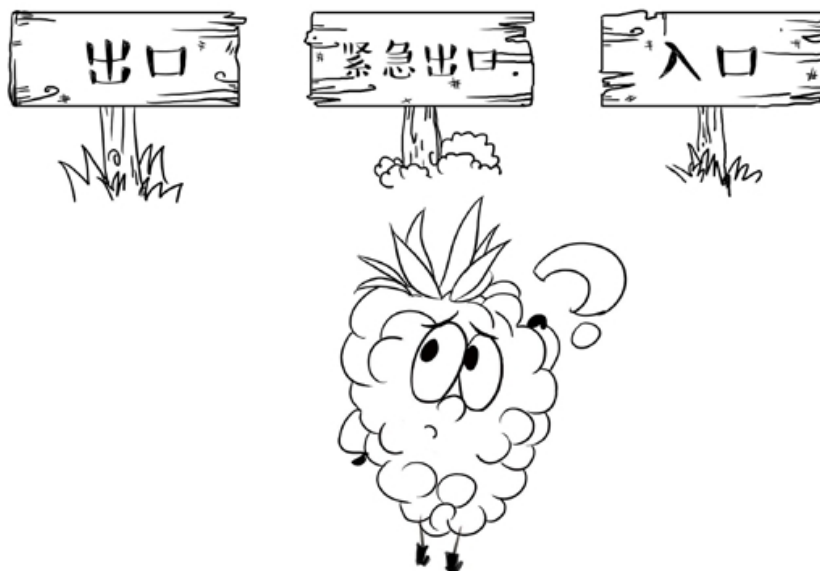


图17-2 入口、出口、紧急出口

我们以bash进程为例，说明三个流的功能。一个运行的bash就是一个进程。默认情况下，bash的标准输入连接到键盘，标准输出和标准错误都连接到屏幕。对于一个程序来说，虽然它总会打开这三个流，但是它会根据需要使用，并不是一定要使用。

想象一下在bash中输入文本的过程。比如敲击键盘，在命令行输入“abc”。键盘的输入成为一个文本流，它通过bash进程的标准输入端口进入bash。bash拿到输入后，不仅进行内部处理，还会把相同的字符输出到标准输出。bash的标准输入最终显示在屏幕上，成为我们在终端看到的“abc”字符。

之前介绍的大部分命令都利用了三大文本端口。比如显示目录内容的ls命令，它获得当前路径下的文件名后，会把这些文件名合成一段文本，用标准输出端口来打印在终端。再比如，设定密码的passwd命令会通过标准输入端口来获得用户输入的密码。如果程序有错误信息，那么错误信息会通过标准错误端口输出，比如删除一个不存在的文件：

```
$rm none-exist-file
```

进程将通过标准错误端口输出文本：

```
rm: none-exist-file: No such file or directory
```

17.3 重新定向

当bash执行一个命令时，这个bash会创建一个子进程用于命令的运行。默认情况下，由于子进程的标准输出与bash相同，因此输出内容出现在bash窗口。如果能让文本流流到文件，而不是显示在屏幕上，那么我们可以利用**重新定向**（redirect）的机制。比如将ls命令输出的文本流导入一个文件：

```
$ls > output.log
```

这里的>符号重新定向了ls的标准输出。标准输出的文本流不再出现在bash窗口中，而是有序地存储于目标文件 **output.log** 中。计算机会新建一个 **output.log** 文件，并将命令行的标准输出指向这个文件。在这个过程中，文本流就像火车换轨，走向不同的方向。

另一个符号>>也可以重新定向，例如：

```
$ls >> a.txt
```

>>符号的作用也是重新定向标准输出。如果 *a.txt* 不存在，那么>>符号的行为和>符号相同，都是新建 *a.txt* 文件，并把文本流导入。但如果 *a.txt* 已经存在，ls产生的文本流会附加在 *a.txt* 的结尾，而不会像>那样每次都新建 *a.txt*。

单一的>和>>符号只会重新定向标准输出。如果标准错误有端口输出，那么输出内容依然按照默认情况，输出到bash窗口。如果想重新定向标准错误，那么可以使用：

```
$rm none-exist-file 2> error.log
```

这里的2代表了标准错误。因此，标准错误重新定向到了文件 *error.log*。你可以分别把标准输出和标准错误重新定向到不同的目的地：

```
$ls 1> output.log 2> error.log
```

1代表了标准输出。符号&>可以同时把标准输出和标准错误指向同一文件：

```
$ls &> output_error.log
```

我们可以用<符号来改变标准输入的来源。比如grep命令，它可以检查一段文本流中是否含有特定的文本。如果只是使用grep命令，那么它将等待键盘输入：

```
$grep abc
```

如果输入的一行字包含“abc”，那么grep将把这一行输出到标准输出中。我们可以重新定向grep的标准输入，让输入内容来自文件而不是键盘：

```
$grep abc < content.txt
```

content.txt中包含了以下内容：

```
abcd  
efgh
```


包含abc的那一行将被输出：

```
abcd
```

当然，我们在重新定向标准输入的同时，还可以重新定向标准输出和标准错误：

```
$grep abc < content.txt &> output.txt
```

17.4 管道

重新定向是把一个进程的标准输出写入文件。**管道**（pipe）也是变更文本流的方向。不过，管道的目的地是另一个进程。借用管道，我们可以把一个进程的输出变成另一个进程的输入。这样，我们可以用管道把两个或者更多命令连接在一起，从而让它们像流水线一样连续工作，不断地处理文本流。在bash中，我们用|表示管道。

```
$echo Hello | grep lo
```

命令echo的功能是把作为参数的文本输出到标准输出。管道把echo的输出导入grep命令。这里的grep命令是从文本流中寻找“lo”字符串。由于输入的“Hello”中包含了“lo”，所以grep命令会打印出“Hello”。

Linux的各个命令实际上高度专业化，并相互独立，每个命令都只专注于一个小的功能。但通过管道，我们可以将这些功能合在一起，实现一些复杂的目的。比如，我们想从一个文件中找出所有包含文本“Tom”的行，并按照字母表顺序排列。文件 *input.txt* 内容如下：

```
2017-06-01 Tom
2017-01-12 Nichole
2017-02-24 Tom
```

想要实现上面的功能，只需要简单的一行：

```
$grep Tom < input.txt | sort
```

这里使用了两个命令。一个是用于寻找文本的grep命令，找到包含“Tom”的各行。这些行通过管道传给sort。命令sort可以给输入的文本流按照行排序。因此，复杂的功能就通过简单命令的组合实现了。

我们还可以把更多的管道连接起来，比如：

```
$ls | grep txt | wc -l
```

命令wc代表“word count”。这个命令用于统计文本中的行、词，以及字符的总数。加上l选项后，wc命令可以统计文本流中总的行数。作为起点，ls命令首先返回当前目录下的文件名，每个文件名占一行。命令grep收到ls输出的文本流，从中抓取包括了“txt”的行。最后，wc命令用于统计grep命令输出的行数。总的来说，这一串命令可以发现当前目录中名字包含了“txt”的文件的总数。

17.5 文本相关命令

Linux中很多命令可以读取文件，把这些文件的内容输出到标准输出。比如有一个文件如下：

```
long_night.txt
It's a long night.
I am far from home.
Home,
Home,
My sweet home.
```

输出整个文件时，可以使用cat命令：

```
$cat long_night.txt
```

命令head和tail分别从文件的开始和结尾输出。比如输出文件开头的3行：

```
$head -3 long_night.txt
```

又如输出文件末尾的两行：

```
$tail -2 long_night.txt
```

此外，还可以用diff命令，只输出两个文件不同的部分：

```
$diff file1 file2
```

file1 和 *file2* 是两个文件的文件名。

上述的文件输出命令，以及输出参数的echo命令，经常作为生成文本流的起点。有了文本流，我们就可以用管道连接起多个命令，从而对文件内容进行编辑。

```
$cat long_night.txt | sort | uniq > another_night.txt
```

命令uniq用于删除一行之后的重复行。上面排列了 *long_night.txt* 的内容，并删除了重复行。编辑后的文本流写入文件 *another_night.txt* 中。

本章介绍了文本流。文本流是UNIX系统的一大特色，Linux继承了这一特色。文本流统一了文件和进程之间交流的接口，从而让程序之间的合作变得更加便利。

[1] 参考附录A。

第18章 我的地盘我做主

Linux是一个多用户系统。多个用户可以同时登录同一台Linux电脑，同时使用，互不干扰。因此，我们必须考虑到用户隐私和用户权限的问题。Linux从UNIX继承来一套用户系统，这套用户系统通过用户权限的设置，可以有效地保护用户隐私，并防止用户进行越权操作。

18.1 我是谁

Linux用户登录时，输入了自己的用户名和密码。用户名是一串可读的文本，比如“pi”。作为惯例，用户名第一位是一个英文字母，后面可以跟随一串英文字母、数字或符号“-”。

如果用户登录通过，那么操作系统就确认了用户的身份。此后用户都以该身份在系统内活动。你可以通过下面的命令找出自己的身份：

```
$who am i
```

从这个命令的返回中，可以看到自己的用户名和最近一次登录时间。

命令who可以返回所有的登录用户：

```
$who
```

如果用户lvor和用户anna都已经登录系统，那么命令将返回：

```
lvor pts/0      2017-11-11 00:00 (180.169.01.01)
anna pts/1      2017-11-11 00:05 (180.169.01.05)
```

在返回的结果中，pts说明了用户登录的终端号。如果用户是用SSH之类的方式远程登录，那么括号中的IP地址说明了用户是从哪个IP地址远程登录的。

在Linux中，我们可以用文本形式的用户名来指代一个用户。比如，命令write可以用来给同一Linux下的其他用户发信息。用户anna发信息给用户lvor：

```
$echo "Where is your draft?" | write lvor
```

命令echo后面的文本用双引号包裹起来，这是为了提醒echo整个双引号内的文本是一个完整的文本，把它作为单一的参数，以防命令错误地根据空格分割为多个参数。

用户不仅是一个单一个体，同时还是一个**用户组**（Group）的成员。组是多个用户的集合，组内的用户享有某些共同的权限。一个用户至少属于一个用户组，可以用groups命令来查找用户所属的组：

```
$groups
```

将返回自己所属的组。

```
$groups anna
```

将返回用户anna所属的组。

用户可以通过文本形式的用户名记住每个用户。不过，在机器底层，会用一个数字代表用户身份。一个用户首先是一个唯一个体。在操作系统眼中，用户个体可以用一个数字，即**用户 ID**（User ID，UID）来表示。组同样可以用一个**数字组 ID**（Group ID，GID）来表示。我们可以用id命令来找到用户的UID和GID：

```
$id pi
```

将返回用户pi的UID和GID：

```
uid=1000(pi) gid=1000(pi) groups=1000(pi)
```

18.2 root和用户创建

除了之前一直在登录使用的pi用户，我们还可以创建其他用户。创建用户的操作需要root权限。在Linux系统中，有一个特殊的root用户，是系统中的神级用户，拥有非常高的权限。root就是上帝，如图18-1所示。通常来说，root账户是由系统管理员掌握的。如果知道root用户的密码，那么可以使用su命令来切换到root用户：

```
$su -
```

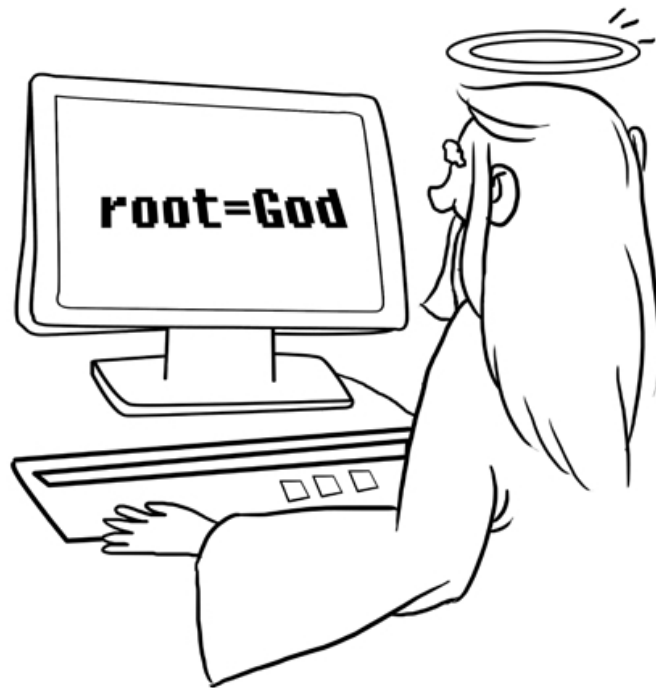


图18-1 root就是上帝

用户root可以做很多普通用户做不到的事，比如监听1024以下的端口、改变文件的拥有者等。由于root权限很高，用户应该避免直接使用root账户进行操作。直接以root权限执行命令式，很容易产生不可挽回的误操作，比如删除根目录：

```
$rm -r /
```

普通用户无权执行该命令，但root用户有权直接执行这一操作，把根目录删除。为了避免类似的灾难，Linux系统引入了sudo。如果普通用户有权执行sudo，那么他可以使用sudo来以root身份执行命令。由于sudo是临时性地扩张用户权限，误操作的概率会大为减小。我们以用户pi为例：

```
$cat /etc/shadow
```

Shell会提示禁止查看。如果以sudo运行相同的命令：

```
$sudo cat /etc/shadow
```

输入pi账号的密码。因为pi有权进行sudo，所以cat命令会以root的身份执行，上面的命令将打印 */etc/shadow* 的内容。

现在，我们可以创建新用户：

```
$sudo adduser tommy
```

命令adduser不仅可以创建用户，还可以帮助用户进行其他的设置，比如为用户建立用户目录、选定用户默认登录Shell等。在创建用户的同时，系统还会创建同名的用户组，用户会被加入该用户组。

我们可以用su命令把自己切换成用户tommy：

```
$su tommy
```

删除用户时，可以使用：

```
$sudo deluser --remove-home tommy
```

也可以用命令来创建用户组：

```
$sudo groupadd genius
```

删除用户组：

```
$sudo groupdel genius
```

18.3 用户信息文件

Linux的用户信息保存在文件 */etc/passwd* 中。通过这个文件，你可以对操作系统中的用户和组进行总览。我们之前对用户的操作，本质上也

是在修改 */etc/passwd* 文件。

在文件 */etc/passwd* 中，每一行代表一个用户。每一行用冒号分为7个部分：

用户名:密码:UID:GID:描述:用户目录:登录 Shell

以用户pi的记录为例：

```
pi:x:1000:1000:./home/pi:/bin/bash
```

在这条记录中描述部分空缺，密码部分用“x”表示。这个符号的含义是，密码以密文的形式保存在文件 */etc/shadow* 中。当然，密码也可以以明文的形式写在 */etc/passwd* 的记录中，但这样系统的安全性会大打折扣。

大多数情况下，用户会有一个属于自己的用户目录，用于存放自己的文件。登录时，bash的当前工作目录，通常会设置成该用户目录。用户root的用户目录在 */root* 下，而普通用户的目录都位于 */home* 下。根据 */etc/passwd* 的记录，用户pi的用户目录是 */home/pi*。

最后的 */bin/bash* 说明了登录之后默认使用的Shell。 */bin/bash* 是bash的程序文件。我们看到有些行的记录使用的Shell是nologin或者false。命令nologin会拒绝登录，而命令false则什么都不做。因此，这些用户没法像普通用户一样，通过Shell来操纵操作系统，因此被称作伪用户。为了系统管理的方便，操作系统创建了这些用户。很多程序会以伪用户的身份运行，以便享有对应的权限。以用户mail为例：

```
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
```

当操作系统调用电子邮件相关程序时，就会用到该伪用户。

同样的，用户组的信息也都保存在 */etc/group* 文件中。这个文件的每一行代表了一个组。每一行用冒号分割成了4段信息。

组名:组密码:GID:用户列表

我们已经了解了组名和GID。组密码并不常用，通常设置成“x”。用户列表用逗号分开，包括了属于该组的全部用户。

18.4 文件权限

Linux系统中的数据保存为文件。因此文件的权限分配就显得异常重要。用户希望自己的某些数据内容能获得隐私保护，用户yutian当然不希望用户anna看到自己存在电脑上的情书。关系到操作系统运行的配置文件不能随意更改。如果每个人都可以写入 */etc/passwd* 这个文件，那么谁都可以随意地创建或删除用户，这将导致整个操作系统十分混乱。因此，操作系统有必要根据用户身份，控制其读、写、执行文件的权限。

在Linux系统中，文件的附加信息包含了权限信息。用ls命令查询文件详情：

```
$ls -l file.txt
```

返回结果可以分为5个部分：

第 1 部分： -rw-r--r--

文件的类型和权限

第 2 部分： 1

文件的链接数

第 3 部分： pi

文件的拥有者和拥有组

第 4 部分： 614

文件的大小，单位是字节。因此，该文件为 614 字节

第 5 部分： Feb 01 22:00

上一次修改文件的时间

文件的权限由第1部分和第3部分控制。第1部分包含了10个字符，第一个字符表示文件类型，和file命令查询结果一致。表示文件权限的是“rw-r--r--”。9个字符分为三组，“rw-”“r--”“r--”，分别对应**拥有者**（Owner）、**拥有组**（Owner Group）和**其他人**（Other）。这是系统用户的3个分类。无论是哪一种分类，都规定了该类是否有读、写、执行的权限。

第一组权限是“rw-”，它表示，如果当前操作用户是文件的拥有者，那么他对该文件就有读取“r”（read）和写入“w”（write）的权限，但符号“-”表示该拥有者无权执行该文件。如果拥有执行权限，则第三位应该是表示执行的“x”字符。以此类推，第二组权限表示，如果用户属于文件的拥有组，那么用户享有读取的权限。第三组表示，任何一个用户都有读取文件的权限。

尽管9位的权限可以任意设置，但通常来说，拥有者享有最多权限，拥有组次之，其他用户的权限最少，如图18-2所示。通过这样一个三级权限系统，每个文件可以有一个联系最密切的用户，即文件的“拥有者”，拥有者对文件享有最高权力。此外，还有一个组的用户区别于系统中的其他用户，对文件拥有特权。由于每个文件都可以把整个系统的用户分成三类，而每一类都可以有不同的权限，所以Linux系统可以非常灵活地设置文件的权限。

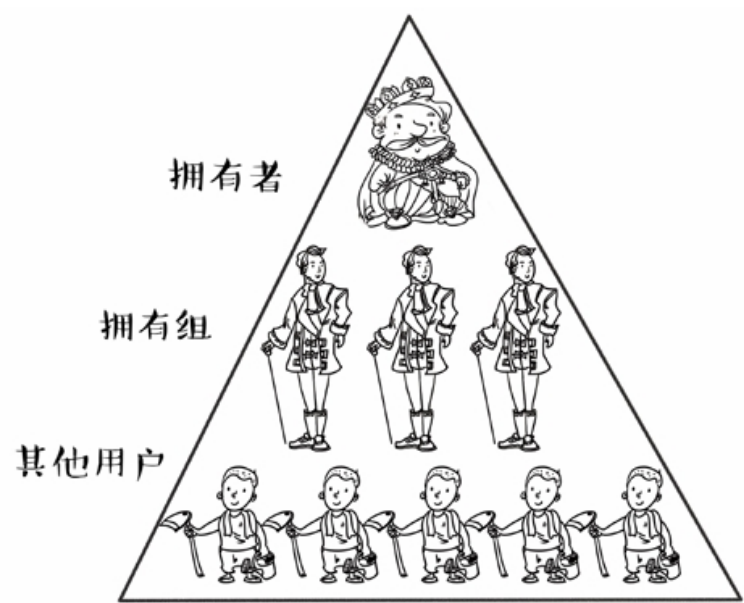


图18-2 拥有者、拥有组、其他用户

下面做一个小练习。如果文件 *sketch.jpg* 的权限标志是：

rwXrw-r--

该文件的拥有者是root，拥有组是vamei，那么对于下面三个用户来说，分别有权执行哪些操作，如表18-1所示。

表18-1 三个用户

	用户名	组
用户 1	pi	pi
用户 2	vamei	vamei
用户 2	yutian	vamei

18.5 文件权限管理

了解了文件权限，我们就可以对文件权限进行设置。你可以从两个方面来控制用户操作文件的权利。一方面，你可以修改文件的拥有者或拥有组，从而重新给用户分类。另一方面，你也可以更改文件的权限标志，赋予每一类用户新的权限。

我们可以用chown命令来改变文件的拥有者和用户组：

```
$sudo chown anna:anna file.txt
```

该命令把文件 *file.txt* 的拥有者改为用户anna，把文件的拥有组改为anna组。

用chmod命令来改变文件的权限标志：

```
$chmod 755 file.txt
```

你必须是文件 *file.txt* 的拥有者或者以root用户的身份才能运行该命令：

```
$sudo chmod 755 file.txt
```

更改之后，文件权限变为：

```
rwXr-Xr-X
```

在命令chmod中，我们用3个数字，如444来对应三类用户的权限。第一个数字代表拥有者权限，第二个数字代表拥有组权限，第三个数字代表其他用户权限。Linux规定，4为读取权，2为写入权，1为执行权。由于第一位7是4、2、1的和，所以拥有者有读、写、执行三项权利。第二

位和第三位的5是4、1的和，因此后面两类用户都有读和执行的权利。你可以尝试一下用444、744和554，看看文件的权限有什么变化。

我们之前提到过，目录也是一个文件。而删除文件这样的操作，实际上是通过写入上级目录文件来实现的。理论上说，控制目录的读写权，也可以控制用户在该目录中增加和删除文件的权利。需要注意的是，由于路径的解析需要对沿途的目录有执行权限，因此一个用户只有对目录文件享有执行权，才能在该目录中增删文件。此外，用户想用cd命令切换工作目录，同样要对该目录有执行权。

本章主要介绍了Linux的用户和权限系统。Linux以用户身份和组身份来管理用户。对于一个文件来说，它给自己的拥有者、拥有组和其他人规定了不同的读、写、执行权限。通过这一机制，Linux可以实现复杂的文件授权。

第19章 会编程的bash（上）

bash是一个命令解释器。在前面章节中介绍了在bash中输入命令，它会把输入的命令转化为特定的动作。本章将介绍bash的可编程性。bash提供了某些类似于C语言的编程语法，从而允许你用编程的方式，来组合使用Linux系统。

19.1 变量

正如我们在C语言中看到的，变量是内存中的一块空间，可以用于存储数据。我们可以通过变量名来引用变量中保存的数据。借助变量，程序员可以复用出现过的数据。bash中也有变量，但bash的变量只能存储文本。

1. 变量赋值

bash和C类似，同样用赋值符号“=”来表示赋值，比如：

```
$var=World
```

赋值就是把文本World存入名为var的变量。根据bash的语法，赋值符号的左右不留空格。赋值符号右边的文本内容会存入赋值符号左边的变量中。

如果文本中包含空格，那么可以用单引号或双引号来包裹文本。用单引号来包裹文本，比如：

```
$var='abc bcd'
```

用双引号来包裹文本，比如：

```
$var="abc bcd"
```

在bash中，可以把一个命令输出的文本直接赋予一个变量：

```
$now=`date`
```

借助``符号，date命令的输出存入了变量now。

还可以把一个变量中的数据赋值给另一个变量：

```
$another=$var
```

用户也可以直接向bash输入数据，这要用到read命令。该命令运行后，bash等待用户输入，比如：

```
$read name
```

命令read后面跟着的name，说明了等待存储数据的变量名。用键盘输入：

```
Vamei
```

然后按Enter键，键盘输入的文本会赋值给变量name，打印变量name检查：

```
$echo $name
```

我们可以看到，变量name中存储的已经是刚才输入的文本“Vamei”了。

2.引用变量

我们可以用\$var的方式来引用变量。在bash中，所谓的引用变量就是把变量翻译成变量中存储的文本，比如：

```
$var=World  
$echo $var
```

打印World，即变量中保存的文本。

在bash中，还可以在一段文本中嵌入变量。bash也会把变量替换成变量中保存的文本，比如：

```
$echo Hello$var
```

文本将打印出HelloWorld。

为了避免变量名和尾随的普通文本混淆，我们也可以换用`${}`的方式来标识变量，比如：

```
$echo $varIsGood
```

因为bash中并没有varIsGood这个变量，所以bash将打印空白行。但如果将命令改为：

```
$echo ${var}IsGood
```

bash通过`${}`识别出变量var，并把它替换成数据，最终echo命令打印出WorldIsGood。

在bash中，为了把一段包含空格的文本当作单一参数，需要用到单引号或双引号，可以在双引号中使用变量，比如：

```
$echo "Hello $var"
```

将打印出Hello World。因为bash会忽视单引号中的变量引用，所以单引号中的变量名只会被当作普通文本，比如：

```
$echo 'Hello $var'
```

将打印出Hello \$var。

19.2 数学运算

在bash中，由于数字和运算符都被当作普通文本，因此无法像C语言一样便捷地进行数学运算，比如执行下面的命令：

```
$result=1+2
$echo $result
```

bash并不会进行任何运算，它只会打印文本“1+2”。

在bash中，还可以通过 $\$(())$ 语法来进行数值运算。在双括号中可以放入整数的加、减、乘、除表达式，bash会对其中的内容进行数值运算，比如：

```
$echo $((2 + (5*2)))
```

它将打印运算结果12。此外，在 $\$(())$ 中，也可以使用变量，比如：

```
$var=1
$echo $(( $var + (5*2) ))
```

打印运算结果11。

可以用bash实现多种整数运算。

- 加法： $\$((1+6))$ ，结果为7。
- 减法： $\$((5-3))$ ，结果为2。
- 乘法： $\$((2*2))$ ，结果为4。
- 除法： $\$((9/3))$ ，结果为3。
- 求余： $\$((5\%3))$ ，结果为2。
- 乘方： $\$((2**3))$ ，结果为8。

现在，你就可以把数学运算结果存入变量了。

```
$result=$(( 1 + 2 ))
```

我们看到，bash支持多种多样的运算符。当一个表达式中有多个算术操作时，就必须考虑算术优先级。bash的算术优先级和数学中的算术优先级类似。优先级从高到低排序如下所示。

(1) 乘方。

(2) 乘法、除法和求余。

(3) 加法和减法。

当优先级相等时，bash按照从左向右的顺序来进行运算。当然，像数学中那样，括号中的内容将优先执行，例如：

```
$echo $((2 + 5*2**(5-3)/2))
```

bash会先执行括号中的减法，然后依次执行乘方、乘法、除法和加法。

19.3 返回代码

在Linux中，每个可执行程序会有一个整数的返回代码。按照Linux的惯例，当程序正常运行完毕并返回时，将返回整数0。因此，C程序中返回0的语句，都出现在C程序中main函数的最后一句。例如下面的foo.c程序：

```
int main(void) {  
    int a;  
    int b;  
    int c;  
  
    a = 6;  
    b = 2;  
    c = 6/2;  
  
    return 0;  
}
```

这段程序可以正常运行。因此，它将在最后一句执行return语句，程序的返回代码是0。在Shell中，运行程序后，可以通过\$?变量来获知返回码，比如：

```
$gcc foo.c
$./a.out
$echo $?
```

如果一个程序运行异常，那么这个程序将返回非0的返回代码，比如删除一个不存在的文件：

```
$rm none_exist.file
$echo $?
```

在Linux中，可以在一行命令中执行多个程序，比如：

```
$touch demo.file; ls;
```

在执行多个程序时，我们可以让后一个程序的运行参考前一个程序的返回代码。比如，只有前一个程序返回成功代码0时，才让后一个程序运行：

```
$rm demo.file && echo "rm succeed"
```

如果rm命令顺利运行，那么第二个echo命令将执行。

还有一种情况是等前一个程序失败了，才运行后一个程序，比如：

```
$rm demo.file || echo "rm fail"
```

当rm命令失败时，第二个echo命令才会执行。

19.4 bash脚本

多行的bash命令写入一个文件就形成了所谓的bash脚本。当bash脚本执行时，Shell将逐行执行脚本中的命令。

1.脚本的例子

用文本编辑器编写一个bash脚本hello_world.bash：

```
#!/bin/bash

echo Hello
echo World
```

脚本的第一行说明了该脚本使用的Shell，即/bin/bash路径的bash程序。脚本正文是两行echo命令。运行脚本的方式和运行可执行程序的方式类似，都是：

```
$/hello_world.bash
```

需要注意的是，如果用户不具有执行bash脚本文件的权限，那么他将无法执行bash脚本。此时，用户必须更换文件权限，或者以其他身份登录，才能执行脚本。

当脚本运行时，两行命令将按照由上至下的顺序依次执行。Shell将打印两行文本：

```
Hello
World
```

bash脚本是一种复用代码的方式。我们可以用bash脚本实现特定的功能。由于该功能记录在脚本中，因此可以通过重复运行同一个文件来实现相同的功能，而不是每次想用的时候都要重新输入一遍命令。我们看一个简单的bash脚本hw_info.bash，它将计算机的信息存入名为 *log* 的文件中：

```
#!/bin/bash

echo "Information of Vamei's computer:" > log
lscpu >> log
uname -a >> log
free -h >> log
```

在bash代码中，可以增加注释。注释以文字的形式在源代码中解释代码功能。注释不会影响代码的运行结果，但可以让代码变得更易读，也更容易维护。在bash脚本中，可以在行首用“#”符号来表示该行是注释。比如在上面的脚本中增加注释：

```
#!/bin/bash
# 输出说明文字
echo "Information of Vamei's computer:" > log
# 输出硬件信息
lscpu >> log
uname -a >> log
free -h >> log
```

2.脚本参数

和可执行程序类似，bash脚本运行时，也可以携带参数。这些参数可以在bash脚本中以变量的形式使用，比如test_arg.bash：

```
#!/bin/bash

echo $0
echo $1
echo $2
```

在bash中，可以用\$0、\$1、\$2.....的方式来获得bash脚本运行时的参数，我们用下面的方式运行bash脚本：

```
$/test_arg.bash hello world
```

\$0是命令的第一部分，也就是./test_arg.bash。\$1代表了参数hello，而\$2代表了参数world。因此，上面的程序将打印：

```
./test_arg.bash
hello

world
```

变更参数，同一段脚本将有不同的行为。这大大提高了bash脚本的灵活性。在上面的hw_info.bash脚本中，我们把输出文件名写成固定的 *log*。我们现在可以修改hw_info.bash脚本，用可变的参数作为输出文件的文件名：

```
#!/bin/bash

echo "Information of Vamei's computer:" > $1
lscpu >> $1
uname -a >> $1
free -h >> $1
```

借助参数可以自由地设置输出文件的名称：

```
$/hw_info.bash output.file
```

3.脚本的返回代码

和可执行程序类似，脚本也可以有返回代码。按照惯例，脚本正常退出时返回代码0。在脚本的末尾，可以用exit命令来设置脚本的返回代码。我们修改hello_world.bash：

```
#!/bin/bash

echo Hello
echo World
exit 0
```

其实在脚本的末尾加一句exit 0并不必要。一个脚本如果正常运行完最后一句，会自动返回代码0。在脚本运行后，可以通过\$?变量查询脚本的返回代码：

```
$/hello_world.bash
$echo $?
```

如果在脚本中部出现exit命令，那么脚本会直接在这一行停止，并返回该exit命令给出的返回代码，比如下面的demo_exit.bash：

```
#!/bin/bash

echo hello
exit 1
echo world
```

你可以运行该脚本，检查其输出结果，并查看返回代码。

19.5 函数

在bash中，脚本和函数有很多相似的地方。脚本实现了一整个脚本文件的程序复用，而函数复用了脚本内部的部分程序。一个函数可以像脚本一样包含多个指令，用于说明该函数如果被调用会执行哪些活动。在定义函数时，我们需要用花括号来标识函数包括的部分。

```
#!/bin/bash

# 定义函数 my_info
function my_info (){
    lscpu >> log
    uname -a >> log
    free -h >> log
}

# 调用函数
my_info
```

脚本一开始定义了函数my_info，my_info是函数名。关键字function和花括号都提示了该部分是函数定义。因此，function关键字并不是必须的。上面的脚本等效于：

```
#!/bin/bash

my_info (){
    lscpu >> log
    uname -a >> log
    free -h >> log
}

my_info
```

花括号中的三行命令就说明了函数执行时需要执行的命令。需要强调的是，函数定义只是食谱，并没有转化成具体的动作。脚本的最后一行是在调用函数。只有通过函数调用，函数内包含的命令才能真正执行。调用函数时，只需要一个函数名就可以了。

像脚本一样，函数调用时还可以携带参数。在函数内部，我们同样可以用\$1、\$2这种形式的变量来调用参数。

```
#!/bin/bash

function my_info (){
    lscpu >> $1
    uname -a >> $1

    free -h >> $1
}

# 第一次调用函数
my_info output.file
# 第二次调用函数，使用了不同的参数
my_info another_output.file
```

在上面的脚本中，进行了两次函数调用。函数调用时分别携带了参数output.file和another_output.file。

19.6 跨脚本调用

使用source命令可以实现函数的跨脚本调用。命令source的作用是在同一个进程中执行另一个文件中的bash脚本。比如，有两个脚本my_info.bash和app.bash。脚本my_info.bash中的内容是：

```
#!/bin/bash

function my_info (){
    lscpu >> $1
    uname -a >> $1
    free -h >> $1
}
```

脚本 app.bash 中的内容是：

```
#!/bin/bash

source my_info.bash
my_info output.file
```

运行 app.bash，执行到 source 命令那一行时，就会执行 my_info.bash脚本。在 **app.bash** 的后续部分，就可以使用my_info.bash中的my_info函数。

本章介绍了bash的变量和运算。此外，函数和命令source提供了函数级别和脚本级别的代码复用。

第20章 会编程的bash（下）

我们已经介绍了函数和脚本两种组合命令的方式。这两种方式都可以把多行命令合并起来，组成一个功能单元。函数和脚本复用代码的方式相对机械。本章会介绍选择和循环两种语法结构，这两种语法结构可以改变脚本的运行顺序，从而编写出更加灵活的程序。

20.1 逻辑判断

bash不仅可以进行数值运算，还可以进行逻辑判断。逻辑判断是确定某个说法的真假。我们在生活中很自然地进行各种各样的逻辑判断。比如“3大于2”这个说法，我们会说它是真的。逻辑判断就是对一个说法判断真假。

在bash中，我们可以用test命令来进行逻辑判断：

```
$test 3 -gt 2; echo $?
```

命令test后面跟有一个判断表达式，其中的-gt表示大于，即greater than。因为“3大于2”这一表达式为真，所以命令的返回代码将是0。如果表达式为假，那么命令的返回代码是1：

```
$test 3 -lt 2; echo $?
```

表达式中的-lt表示小于，即less than。

数值大小和相等关系的判断是最常见的逻辑判断。除了上面的大于和小于判断，我们还可以进行以下的数值判断。

- 等于：

```
$test 3 -eq 3; echo $?
```

- 不等于：

```
$test 3 -ne 1; echo $?
```

- 大于等于:

```
$test 5 -ge 2; echo $?
```

- 小于等于:

```
$test 3 -le 1; echo $?
```

bash中最常见的数据形式是文本，因此也提供了很多关于文本的判断。

- 文本相同:

```
$test abc = abx; echo $?
```

- 文本不同:

```
$test abc != abx; echo $?
```

- 按照词典顺序，一个文本在另一个文本之前:

```
$test apple > tea; echo $?
```

- 按照词典顺序，一个文本在另一个文本之后:

```
$test apple < tea; echo $?
```

bash还可以对文件的状态进行逻辑判断。

- 检查一个文件是否存在:

```
$test -e a.out; echo $?
```

- 检查一个文件是否存在，而且是普通文件:

```
$test -f file.txt; echo $?
```

- 检查一个文件是否存在，而且是目录文件:

```
$test -d myfiles; echo $?
```

- 检查一个文件是否存在，而且是软链接：

```
$test -L a.out; echo $?
```

- 检查一个文件是否可读：

```
$test -r file.txt; echo $?
```

- 检查一个文件是否可写：

```
$test -w file.txt; echo $?
```

- 检查一个文件是否可执行：

```
$test -x file.txt; echo $?
```

在做逻辑判断时，可以把多个逻辑判断条件用“与、或、非”的关系组合起来，形成复合的逻辑判断。

```
! expression  
expression1 -a expression2  
expression1 -o expression2
```

20.2 选择结构

逻辑判断可以获得计算机和进程的状态。bash可以根据逻辑判断，让程序有条件地运行，这就是所谓的选择结构。选择结构是一种语法结构，可以让程序根据条件决定执行哪一部分指令。最早的程序都是按照指令顺序依次执行的。选择结构打破了这一顺序，给程序带来更高的灵活性。

最简单的，我们可以根据条件来决定是否执行某一部分程序，比如下面的demo_if.bash脚本：

```
#!/bin/bash

var = `whoami`
if [ $var = "root" ]
then
    echo "You are root"
    echo "You are my God."
fi
```

这个脚本使用了最简单的if结构。关键字if后面跟着[]，里面是一个逻辑表达式。这个逻辑表达式就是if结构的条件。如果条件成立，那么if将执行then到fi之间包含的语句，我们称之为隶属于if的代码块。如果条件不成立，那么隶属于if的代码块不执行。因此，if结构的流程如图20-1所示。

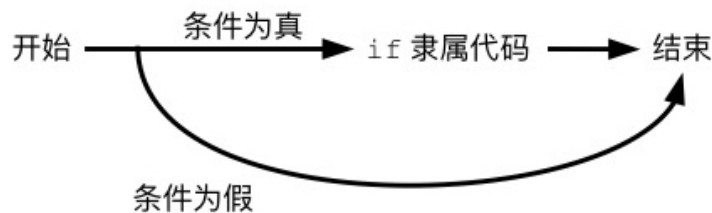


图20-1 if结构

这个例子的条件是判断用户是否为root。因此，如果是非root用户执行该脚本，那么Shell不会打印任何内容。

我们还可以通过if else结构，让bash脚本从两个代码块中选择一个执行。该选择结构同样有一个条件。如果条件成立，那么将执行if附属的代码块，否则执行else附属的代码块，流程如图20-2所示。



图20-2 if else结构的流程

下面的demo_if_else.bash脚本是if else结构的一个小例子：

```
#!/bin/bash

filename=$1
if [ -e $filename ]
then
    echo "$filename exists"
else
    echo "$filename NOT exists"
fi

echo "The End"
```

if后面的“-e \$filename”作为判断条件。如果条件成立，即文件存在，那么执行then后面的代码；如果文件不存在，那么脚本将执行else语句中的echo命令，末尾的fi结束整个语法结构。脚本继续以顺序的方式执行剩余内容。运行脚本：

```
$. /demo_if_else.bash a.out
```

脚本会根据a.out是否存在打印出不同的内容。

我们看到，在使用if...else结构时，我们可以实现两部分代码块的选择执行。而在if代码块和else代码块内部，可以继续嵌套选择结构，从而实现更多个代码块的选择执行。比如脚本demo_nest.bash：

```
#!/bin/bash

var=`whoami`
echo "You are $var"

if [ $var = "root" ]
then
    echo "You are my God."
else
    if [ $var = "vamei" ]
```

```

        then
            echo "You are a happy user."
        else
            echo "You are the Others."
        fi
    fi
fi

```

在bash下，还可以用case语法来实现多程序块的选择执行，比如下面的脚本demo_case.bash：

```

#!/bin/bash

var=`whoami`
echo "You are $var"

case $var in
    root)
        echo "You are God."
        ;;
    vamei)
        echo "You are a happy user."
        ;;
    *)
        echo "You are the Others."
        ;;
esac

```

这个脚本和上面的demo_nest.bash功能完全相同，可以看到case结构与if结构的区别。关键字case后面不再是逻辑表达式，而是一个作为条件的文本。后面的代码块分为三个部分，都以文本标签的形式开始，以;;结束。case结构运行时会逐个检查文本标签。当条件文本和文本标签对应时，bash就会执行隶属于该文本标签的代码块。如果是用户vamei执行该bash脚本，那么条件文本和vamei标签对应上，脚本就会打印：

```

You are a happy user.

```

文本标签除了是一串具体的文本，还可以包含文本通配符。结构case中常用的通配符，如表20-1所示。

表20-1 结构case中常用的通配符

通 配 符	含 义	文本标签的例子	通过的条件文本
*	任意文本	*)	Xyz, 123, ...
?	任意一个字符	a?c)	abc, axc, ...
[]	范围内一个字符	[1-5][b-d])	2b, 3d, ...

在上面的程序中，最后一个文本标签是通配符*，即表示任意条件文本都可以触发此段代码块运行。当然，前提是前面的几个文本标签都没有触发代码运行。

20.3 循环结构

循环结构是编程语言中一种常见的语法结构。循环结构的功能是重复执行某一段代码，直到计算机的状态符合某一条件。在while语法中，bash会循环执行隶属于while的代码块，直到逻辑表达式不成立。比如下面的demo_while.bash：

```
#!/bin/bash

now=`date +%Y%m%d%H%M`
deadline=`date --date='1 hour' +%Y%m%d%H%M`

while [ $now -lt $deadline ]
do
    date
    echo "not yet"
    sleep 10
    now=`date +%Y%m%d%H%M`
done

echo "now, deadline reached"
```

关键字do和done之间的代码是隶属于该while结构的代码块。在while后面跟着条件，该条件决定了代码块是否重复执行下去。这个条件是用当前的时间与目标时间对比。如果当前时间小于目标时间，那么代码块就会重复执行下去。否则，bash将跳出循环，继续执行后面的语句，流程如图20-3所示。

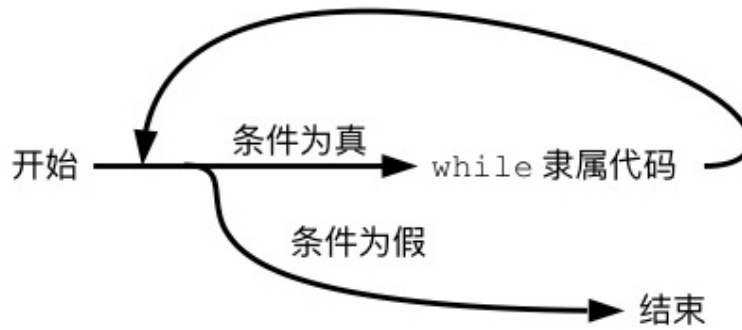


图20-3 while结构的流程

如果while的条件始终为真，那么循环会一直进行下去。下面的程序以无限循环的形式，不断播报时间。

```
#!/bin/bash

while true
do
    date
    sleep 1
done
```

语法while的终止条件是一个逻辑判断。如果在循环过程中改变逻辑判断的内容，那么我们很难在程序执行之前预判循环进行的次数。正如之前在demo_while.bash中看到的，我们在循环进行过程中改变着作为条件的逻辑表达式，不断地更新参与逻辑判断的当前时间。与while语法对应的是for循环。这种语法会在程序进行前确定好循环进行的次数，比如demo_for.bash：

```
#!/bin/bash

for var in `ls log*`
do
    rm $var
done
```

在这个例子中，命令ls log*将返回所有以log为开头的文件名，这些文件名之间由空格分隔。当循环进行时，bash会依次取出一个文件名，赋

值给变量var，并执行do和done之间隶属于for结构的程序块。由于ls命令返回的内容是确定的，因此for循环进行的次数也会在一开始确定下来。

在for语法中，也可以使用自己构建一个由空格分隔的文本。由空格区分出来的每个子文本会在循环中赋值给变量，比如：

```
#!/bin/bash

for user in vamei anna yutian
do
    echo $user
done
```

此外，for循环还可以和seq命令配合使用。命令seq用于生成一个等差的整数序列，命令后面可以跟3个参数，第一个参数表示整数序列的开始数字，第二个参数表示每次增加多少，第三个参数表示序列的终点。因此，下面命令：

```
$seq 1 2 10
```

将返回：

```
1 3 5 7 9
```

可以看到，seq返回的也是由空格分隔开的文本。因此，seq的返回结果也可用于for循环。

结合for循环和seq命令，我们可以解一些有趣的数学问题。比如高斯求和，即计算从1到100的所有整数的和，可以用bash解决。

```
#!/bin/bash
total=0
for number in `seq 1 1 100`
do
    total=$(( $total + $number ))
done

echo $total
```

这个问题还可以用do while循环来求解。

```
#!/bin/bash
total=0
number=1
while :
do
    if [ $number -gt 100 ]
    then
        break
    fi
    total=$(( $total + $number ))
    number=$(( $number + 1 ))
done

echo $total
```

这里break语句的作用是在满足条件时跳出循环。

如果想计算1到100所有不被3整除的数的和，则可以使用continue语句，跳过所有被3整除的数。

```
#!/bin/bash
total=0
for number in `seq 1 1 100`
do
    if (( $number % 3 == 0 ))

    then
        continue
    fi
    total=$(( $total + $number ))
done

echo $total
```

20.4 bash与C语言

到了这里，我们已经介绍完bash语言的基本语法。bash语言和C语言都是编程语言，它们都能通过特定的语法来编写程序，而程序运行后都

能实现某些功能。虽然在语法细节上存在差异，但两种语言都有以下语法。

- 变量：在内存中储存数据。
- 循环结构：重复执行代码块。
- 选择结构：根据条件执行代码块。
- 函数：复用代码块。

编程语言的作者在设计语言时，往往会借鉴已有编程语言的优点，这是不同编程语言具有相似性的一大原因。程序员往往要掌握不止一套编程语言。相似的语法特征，会让程序员在学习新语言时感到亲切，从而促进语言的推广。

bash和C的相似性，也来自于它们共同遵守的编程范式——面向过程编程。支持面向过程编程的语言，一般都会提供类似于函数的代码封装方式。函数把多行指令包装成一个功能。只要知道了函数名，程序可以通过调用函数来使用函数功能，最终实现代码复用。除了面向过程编程，还有面向对象和函数式的编程范式。每种编程范式都提供了特定的代码封装方式，并达到代码复用的目的。值得注意的是，近年来出现的新语言往往会支持不止一种编程范式。

除了相似性，还应该注意bash和C程序的区别。bash的变量只能是文本类型，C的变量却可以有整数、浮点数、字符等类型。bash的很多功能，如加、减、乘、除运算，都是通过调用其他程序实现的，而C程序直接就可以进行加、减、乘、除运算。可以说，C语言是一门真正的编程语言，C程序最终会编译成二进制的可执行文件，CPU可以直接理解这些文件中的指令。

一方面，bash是一个Shell，它本质上是一个命令解释器程序，而不是编程语言。用户可以通过命令行的方式来调用该程序的某些功能。所谓的bash编程，只是命令解释器程序提供的一种互动方法。bash脚本只能和bash进程互动。它不能像C语言一样，直接调用CPU的功能。因此，bash能实现的功能会受限，运行速度也比不上可执行文件。

另一方面，bash脚本也有它的好处。C语言能接触到底层的東西，但使用起来很复杂。有时候，即使已经知道如何用它实现一个功能，写代码依然是一个很烦琐的过程，而bash正相反。由于bash可以便捷地调用已有的程序，因此很多工作可以用数行脚本解决。此外，bash脚本不用编译

就可以由bash进程理解并执行，因此开发bash脚本比写C程序要快很多。Linux的系统运维工作，如定期备份、文件系统等管理，就经常使用到bash脚本。总之，bash编程知识是晋级为资深Linux用户的必要条件。

第21章 完整架构

Linux系统可以分为内核和应用程序两个主要部分，但如果细分，内核和应用程序之间，还可以有更精细的模块划分。完整的Linux系统架构，如图21-1所示，下面分别来看Linux架构中的不同部分。

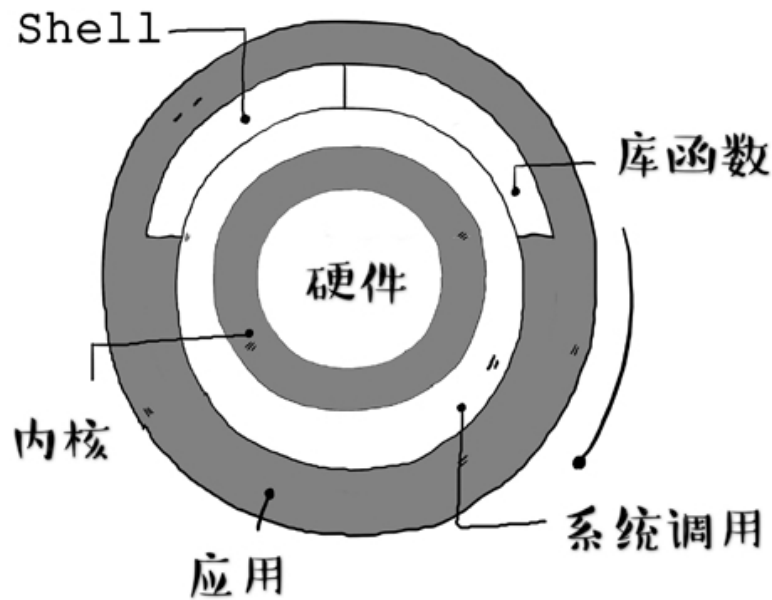


图21-1 Linux系统架构

21.1 内核模式与系统调用

计算机启动之后，Linux的内核程序启动成为一个单一的内核进程。这个单一进程将执行内核的相关功能。内核进程有权调用所有的计算机资源。当应用程序运行时，内核会分配给该应用程序一定的计算机资源。应用程序与硬件之间的互动，也必须经由内核进行。因此，即使是一个应用程序，它的运作也离不开内核。我们把内核程序的活动称为**内核模式**（Kernel Mode），而把应用程序的活动称为**用户模式**（User Mode）。

应用程序可以通过特定的接口来调用内核功能。用户单次的内核调用，可以称为一次**系统调用**（System Call）。接口中的系统调用有大约两百种，每种系统调用都有特定的名称和调用方式。在Shell中输入下面的命令就可以查看Linux下所有的系统调用。

```
$man 2 syscalls
```

在Linux最常见的开发语言C中，系统调用都制作成了有特定函数名的函数。你可以通过：

```
$man 2 read
```

来查看系统调用read这一系统调用的说明。命令中的2，对应了系统调用类相关的查询。命令man定义的几个查询类可以通过下面的命令查看：

```
$man man
```

常见的系统调用如下。

- read，文件读取。
- write，文件写入。
- fork，复制当前进程。
- wait，等待某个进程完成。
- chdir，改变工作目录。

每次系统调用，用户程序就触发了内核的特定动作。以bash中的内置函数cd为例。bash实际上执行的就是进行chdir这个系统调用。系统调用发生后，作为用户进程的bash暂停，操作系统转入内核模式。当内核完成chdir的系统调用后，即更改了进程的工作目录，bash进程将恢复执行，程序又重回用户模式。

对于用户程序来说，系统调用是内核的最小功能单位。用户程序不可能调用超越系统调用的内核动作。一个系统调用就像是汉字的一个笔画。任何一个汉字都要由基本的笔画构成，没有人可以臆造笔

画。通过系统调用这个接口，Linux实现了内核封装，隐藏了底层的复杂性，也提高了上层应用的可移植性。

21.2 库函数

系统调用提供的功能非常基础，使用起来很麻烦。一个给变量分配内存空间的简单操作，就需要动用多个系统调用。为了方便，程序员还可以使用上层的**库函数**（Library Function），来实现特定的“组合拳”。

函数是面向过程语言中复用代码功能的一种方式。所谓的“库”是一个文件，它包含了多个常用函数。在C语言中，我们可以跨文件地调用库中的函数。C语言本身就规定了**C标准库**（C Standard Library）。之前使用的printf函数，就属于C标准库：

```
#include <stdio.h>

int main()
{
    printf("Hello world");
    return 0;
}
```

如果只是使用系统调用来实现上面的程序，那么我们编程的工作量就会增加：

```

#include<unistd.h>
#include<fcntl.h>

int main (void)
{
    const char msg[] = "Hello world";
    int length;
    length = sizeof( msg ) - 1
    write( STDOUT_FILENO, msg, length);

    return 0;
}

```

在这个程序中，我们需要手动为文本分配内存空间，并计算文本的长度。而在之前的程序中，这些工作都由printf代劳。此外，printf还可以改变文本输出的格式，并且优化输入和输出的效率，这些都是write系统调用所不具备的。

我们可以用man来查阅库函数的帮助文档。库函数的查询类是3：

```
$man 3 printf
```

除了C标准库，Linux系统中还有很多其他的库，比如POSIX标准库。UNIX操作系统都会安装POSIX标准库。函数malloc就是POSIX标准库中的库函数，这个函数常用于内存分配。目录 **/lib** 和 **/lib64** 下存放着Linux系统自带的库。用户也可以在目录 **/usr/lib** 下安装额外的库。这些库函数把程序员从细节中解救出来，极大提高了编程效率。

21.3 Shell

系统调用和库函数是为程序员准备的，普通用户使用的都是编译好的应用程序。在所有应用程序中，Shell的地位相当特殊。在历史上，Shell曾经是Linux用户使用计算机的唯一界面。用户必须在Shell中用命令的方式来运行程序。当然，随着图形化桌面的发展，让用户有了一个新的运行程序的界面。但图形化桌面并不能完全取代Shell的地位。很多程序只能通过Shell的方式启动，比如用于下载的wget。

Shell把Linux系统的很多功能开放给用户。对于有编程能力的高级用户来说，他们当然可以通过系统调用、库函数和C语言编程来完整地发挥Linux系统的能力。在另一个极端，大多数应用程序有其专攻的方向，比如文字编辑、收发邮件、显示网页等。一个应用程序只发挥了操作系统很小的一部分能力。Shell介于两者之间，把相对底层的功能用简单而统一的接口呈现给用户，比如用简单的文本符号“|”来使用管道，又如用内置命令cd来改变Shell的工作目录等。对于编程经验有限的普通用户来说，Shell开放出来的系统功能是福音。由于Shell很简单，又容易和其他应用程序互动，它的开放接口也深受资深程序员的喜爱。

我们也已经看到Shell的可编程性。借着这种可编程性，Shell可以充当不同命令之间的“胶水”，把功能专一的应用程序组合成功能多样的脚本。此外，脚本还可以预设一连串的操作。用户可以把耗时的手工操作编写成Shell脚本，让Linux系统按照脚本的指挥自动运行。很多Linux计算机就是凭着Shell脚本，在无人工干预的情况下长期工作。

21.4 用户程序

整个架构的最上层就是应用程序。我们已经知道，应用程序是二进制的可执行文件。在 `/bin` 和 `/usr/bin` 中，我们可以看到不少这样的可执行文件。可执行文件还可以出现在文件系统的其他位置。按照惯例，应用程序所在的目录都被命名为 **bin**。这里的bin是指binary，即二进制。

Linux中大多数可执行文件都是由C语言编写的。当然，你还可以用其他的语言来编写程序，如C++和Fortran。C语言写成的程序是可读的文本，必须经过编译才能生成可执行文件。我们可以用gcc命令把C程序直接编译成可执行文件。但在幕后，gcc实际上要做好几件事。它必须对源代码进行一定的文本处理，把人类可读的文本翻译成机器可读的二进制序列，最后还要找到程序依赖的库文件。

编译成功后，就可以用Shell运行应用程序。两种运行程序的方式如下。

- 直接输入程序名，如ls、man、wget。
- 在应用程序所在的目录中用“./可执行文件名”的方式，比如./a.out。

这两种方式的区别在于，前一类的命令包含在Shell的默认路径中。输入这些命令时，Shell会搜索默认路径，直到找到同名的程序并运行。默认路径在Shell中保存为一个变量，你可以打印出该默认路径：

```
$echo $PATH
```

可以看到，**/bin** 包含在PATH变量中。由于ls程序位于 **/bin** 中，因此我们可以不加路径地运行该程序。如果应用程序所在位置在默认路径之外，那么我们就必须切换工作目录，或者使用完整路径，如：

```
$/home/vamei/a.out
```

Linux系统提供了一个多层次的互动平台。从应用程序到Shell，再到库函数和系统调用，用户可以一层层地接近底层。越往底层，用户受到的限制越少，能发挥的功能越丰富。当然，学习难度也越来越大。对于爱好挑战的计算机爱好者来说，这样的“学习升级”过程也充满了趣味。

第22章 函数调用与进程空间

在Linux中，应用程序位于整个架构的顶层。应用程序的进程会获得一块独立的内存空间，即进程空间。C语言中变量的相关操作实际上就作用于进程空间。应用程序大部分是面向过程的C语言编写的，因此进程空间的使用也受到面向过程思维的影响。这里将用一章的篇幅来讲解进程空间的结构。

22.1 函数调用

函数是面向过程语言提供的抽象语法，也是C语言区别于指令式程序的关键。在程序中，要先定义函数，然后才能调用函数。函数定义中说明了在函数调用发生时，进程应该做些什么事情。我们先以一个C程序为例，深入了解函数调用。

```
#include <stdio.h>

float PI=3.1415926;

float power(float x, int n) {
    float result;
    int i;
    result = 1.0;
    for(i=0; i<n; i++) {
        result = result * x
    }
    return result;
}

float calculate_area (float r) {
    float square;
    square = power(r, 2);
    return PI*square;
}
```

```

void main(void) {
    float area;
    float r;
    r = 1.2;

    area = calculate_area(r);
    printf("Area of the first circle: %6.2f \n", area);

    r = 5.1;
    area = calculate_area(r);
    printf("Area of the second circle: %d \n", area);
}

```

这段程序中出现了一种新的数据类型，即**浮点数**（float），用于存储1.68或3.14这样的浮点数。除此之外，程序中声明了三个函数，power、calculate_area和main。函数有不同的功能，power用于计算一个数的任意次方，calculate_area用于计算一个圆的面积。函数main是主函数，在计算出两个圆的面积之后，把结果打印了出来。下面研究这三个函数，以及它们之间的调用关系。

函数声明的第一行除了说明了函数名，还在一开始说明了函数返回值类型。函数power和函数calculate_area的返回值是浮点类型，而函数main的返回值是整数类型。在讲解bash时，我们提到了每个命令都会返回一个整数值，用来表示函数是否成功运行。命令的返回值，实际上就是命令程序的main函数的返回值。相应的，函数中return语句返回的值要和这个类型声明吻合。以calculate_area的函数定义为例，最开始的float说明了返回值类型。相应的，calculate_area中最后一句return返回的也应该是一个浮点数。

在函数声明中，还说明了函数参数的类型。如果说函数的返回值是函数的输出，那么参数就是它的输入。函数定义的第一行，函数名后的括号中包含了函数的参数列表。函数calculate_area接受一个浮点数作为参数。根据参数列表，该参数名为r。函数内部就可以像使用一个变量那样，通过“r”这个名称使用参数。一个函数可以有多个参数。函数power就有两个参数，第一个参数是浮点数，第二个参数是整数。

下面观察函数调用发生的顺序。main函数调用了两次calculate_area函数。每次调用calculate_area函数时，该函数内部又会调用power函数。上

级函数调用下级函数后，被调用的下级函数就会开始运行。函数的调用过程如图22-1所示。

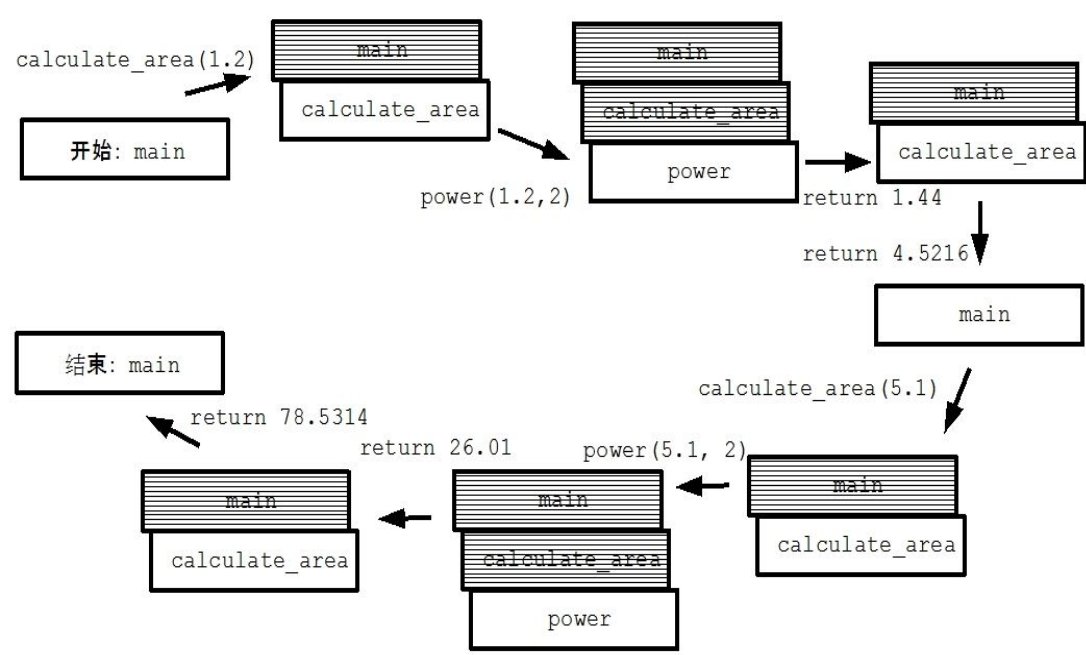


图22-1 函数的调用过程

下级函数运行时，上级函数只是暂停。等到被调用函数返回，上级函数才恢复运行，继续执行下面的语句。在C程序中，main函数总是最早被调用的，因此main函数位于最高级，后面被调用的函数置于下方，但下级函数先执行。除非下级函数运行结束，否则上级函数都处于暂停状态。也就是说，后来的函数将先获得执行。

22.2 跳转

我们可以深入编译后的指令式程序，看看计算机如何在底层实现C语言的函数调用。首先，在进程空间中，有一块名为**程序段**（TEXT）的区域。进程启动后，会先把程序文件加载到进程空间的程序段中。程序文件是编译后的指令式程序。加载到程序段之后，每个指令占据一个存储单元，并可以通过内存地址来定位。随后，计算机将按照指令顺序，依次执行每条指令。

函数中包含了需要依次执行的多个指令。函数指令存储在一块连续的区域中，包含了多条指令。函数也可以通过内存地址来确定位置。这个内存地址就是函数第一条指令的内存地址。为了实现程序复用，每个函数在程序段中只会存一次。表22-1是TEXT区域的示例。需要注意的是，表中的内存地址只是示意，每次编译运行时，具体的内存地址都会有差别。

表22-1 TEXT区域的示例

内存地址	指令内容	归属函数
101	指令 1	square
102	指令 2	
103	指令 3	
.....		
152	指令 1	calculate_area
153	指令 2	
154	调用 power	
.....		
175	return	
.....		
238	指令 1	main
239	指令 2	
.....		
253	第一次调用 calculate_area	
.....		
267	第二次调用 calculate_area	

如果要想实现函数调用的流程，就没法使用指令式的顺序执行。在main函数顺序执行中，遇到calculate_area就不能继续执行自身的下一句指令，必须跳转到calculate_area指令所在的区域。在表22-1的进程空间，就是跳转到152的内存位置。calculate_area函数运行完成后，也必须跳转回上级函数离开的位置。即使是指令式程序，也有跳转的用法。在跳转语句中，只需要说明指令的内存地址，就可以让进程在这个内存地址的位置进行执行。

在进程开始前，程序加载入程序段，每个函数就已经有了确定的内存地址。当函数调用时，进程只需要跳转到函数指令所在的位置就可以了。然而，函数返回时的跳转就变得复杂了。函数调用可能发生在不止一个地方。因此，当被调用函数返回时，应该返回到的指令是不固定的。比如在示例程序中，`calculate_area`函数被调用了两次，分别发生在`main`函数的第4行和第7行。因此，两次函数调用的返回地址应该是不同的。

问题的关键在于，函数调用的某些信息是可变的。为了记录函数调用中的可变信息，进程开辟了另一块名为**栈**（Stack）的内存空间。

22.3 栈与情境切换

栈是为了配合函数调用而产生的。既然如此，栈的组织方式也和函数调用类似。回顾函数调用的逻辑顺序，当函数调用发生时，上级函数暂停，下级函数开始工作。因此，函数调用的逻辑顺序有个特点，总是最下级的函数处于激活状态。

我们对比地看栈的工作方式。在`main`函数运行时，内存中就会有一个对应`main`函数的内存单元出现，用来记录`main`函数的可变信息，比如`main`函数返回时，应该跳转的地址。这个帧就是整个栈的起点。此后，每次有新的函数调用发生时，栈就会向下增加一个帧，对应这一次的函数调用。在创建这个帧时，进程就会记下离开上级函数前的地址，也就是新函数调用的返回地址，所有的帧就组成了一个栈。借助栈的存储能力，函数返回就不再是一个问题了。

图22-2说明了示例程序运行时栈的变化情况。当帧最下方的函数完成时，栈会弹出最下方的帧，取出其中的返回地址，并删除帧的内存空间。进程跳转到返回地址继续运行，原本暂停了的上级函数继续执行。与此同时，暴露在栈最下方的帧恰好对应了恢复激活状态的上级函数。随着函数的调用和返回，栈也不断变化，增加一帧或减少一帧。等到`main`函数也返回时，栈最高级的帧也被删除，整个程序就运行结束了。

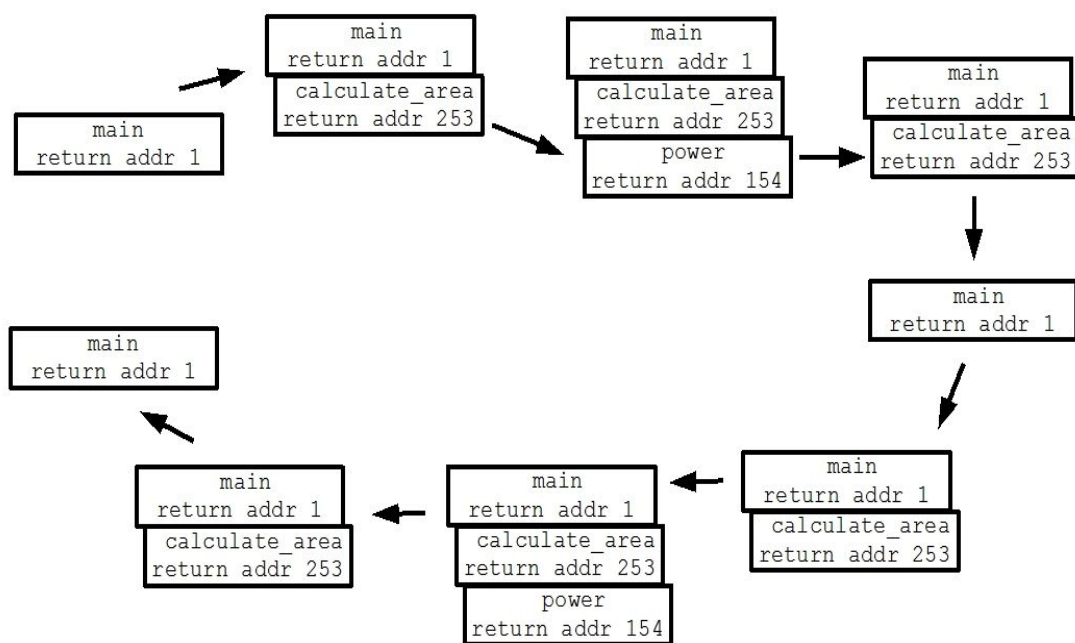


图22-2 栈的变化，以及栈中存的返回地址

栈的变化过程和函数调用的变化过程很类似。这种相似性是个自然的结果。毕竟，帧本身就是配合着函数调用工作的。因此，栈完全符合了函数调用的逻辑顺序：总是最下方、对应当前函数的帧处于活跃状态。

22.4 本地变量

除了返回地址，栈中还能存储其他数据。由于栈伴随着函数调用发生变化，栈中存储的其他数据也跟着函数调用诞生和消失。我们先来看每个帧中存储的**本地变量**（Local Variable）。所谓的“本地”，就是指函数内部。一个本地变量只能在函数内部声明，比如calculate_area函数中的s。当函数被调用时，该函数的本地变量才在对应的帧中出现。当函数调用结束时，帧被清空，其中存储的本地变量自然会被清空。因此，本地变量只能用于存储函数调用相关的数据。

calculate_area函数的目的是计算圆的面积。在计算过程中，我们用本地变量square存储了中间结果，即半径的平方。等到函数结束时，我们已经计算出圆的面积并返回，那么square中存储的中间结果就不重要了。函

数返回时，帧被清空，变量square也伴随着帧从内存中消失，内存空间自然而然地腾了出来。

再来观察power函数。这个函数用于计算一个数的任意次方。函数中的本地变量result同样用于计算中间结果。此外，函数中还有一个本地变量i，用于for循环。我们已经在bash中见过for循环，这种循环结构可以进行固定次数的循环操作，而C语言中的for循环也类似。在进行循环的过程中，变量i记录了当前循环的次数。换句话说，本地变量同样反映了函数当前的状态。power函数的帧中内容，如表22-2所示。

表22-2 power函数的帧中内容

类 型	示 例
返回地址	154
本地变量	result i
参数	x n

本地变量随着函数调用诞生，又随着函数返回消失。形象地说，本地变量只存活在函数内部。当然，如果函数调用了下级函数，上级函数的本地变量依然保持在帧中。不过，C语言只允许函数调用当前帧中的内容。因此，激活函数只能操作当前帧中的内容，没法读取或写入上级帧的本地变量。正是有了这样的机制，本地变量完全封闭到了函数内部。定义本地变量的函数内部，就称为本地变量的作用域。因为各个函数“看不到”其他函数的本地变量，所以不同函数可以使用相同的本地变量名。

除了本地变量，帧中还存储着函数的参数。参数用于存储函数的输入。我们在函数调用时输入的数据，就会放在帧中分配给参数的位置上。由于参数也存活于帧中，因此参数的作用域和本地变量完全相同。事实上，你完全可以像使用本地变量那样在函数内部使用参数，让参数记录函数的中间结果或状态。只不过，这种做法违背了参数的初衷，因此程序员很少会这么做。

22.5 全局变量和堆

图22-3展示了完整的进程空间。

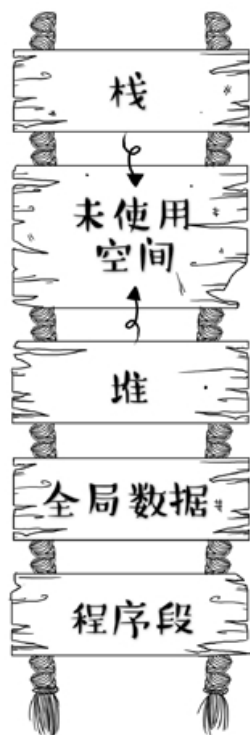


图22-3 进程空间

除了局部变量，进程空间中还有全局变量和动态变量。在内存空间中，**全局数据**（Global Data）部分用于存放**全局变量**（Global Variable）。“全局”这个名字说明了全局变量的作用域，即所有的函数。在任何一次函数调用中，都可以使用全局变量。在C程序中，在函数之外声明的变量就是全局变量，如示例程序中的PI。在calculate_square函数中，就可以直接调用PI。我们也可以在任意函数中给某个全局变量赋值。因为全局变量的修改有可能影响到多个函数，所以修改全局变量很容易造成意想不到的错误。通常来说，全局变量只用于存储不变的常量。

堆（Heap）用于存放**动态变量**（Dynamic Variable）。和全局变量类似，动态变量可以被所有的函数看到。不过，全局变量的个数和类型在程序一开始就是确定的，全局数据区域的大小也是确定的，而动态变量可以在进程中产生和消失。当进程创建动态变量时，堆的区域就会增长，占据更多的内存空间。堆增长的部分就是动态变量的空间。

堆和栈是相互独立的区域，堆的空间不随着函数调用自动增长或清空。在堆的支持下，动态变量的作用域同样是全局。在任意一个函数的内部，都可以通过动态变量的地址来访问动态变量中的数据。每个函数都可以通过malloc系统调用来在堆上创建动态变量。这个系统调用返回的

是动态变量的内存地址。函数之间可以通过参数或返回值来交换该地址，从而跨函数地共享数据，本地变量就无法实现上述功能。

当不再需要某个动态变量时，可以通过free系统调用来释放动态变量占据的内存空间。C语言中的一个常见错误是**内存泄漏**（Memory Leakage），就是指没有释放不再使用的动态变量，导致进程空间的可用内存不足。

本章介绍了函数调用过程和进程空间的结构，两者相辅相成，共同来完成进程的任务。

第23章 穿越时空的信号

如果说操作系统是一栋大楼，那么内核就是这栋大楼唯一的管理人员，应用程序的进程就是大楼里的房客。一般情况下，进程躲在自己的房间里，专注于自己的事情，而不必考虑其他进程。但有的时候，进程也要打破封闭，相互交流。信号就是一种向进程传递信息的方式。

23.1 按键信号

在Shell中可以通过快捷键Ctrl+C来中断正在运行的进程，或者用快捷键Ctrl+Z来中止进程。按下这些按键时，Shell都向进程发出了信号。进程捕捉到这些信号后，会根据信号的含义来执行特定的动作，如结束进程或者暂停。

之前使用Shell时，都是在Shell中输入一个命令，然后等待命令完成。命令完成后，我们才能执行其他命令。在进程运行期间，Shell的命令行输入会**阻塞**（Block）。命令行阻塞之后，Shell不再接受新的命令，比如：

```
$ping localhost > log
```

命令ping的进程占据了整个舞台，因此称为前台进程。每个Shell最多有一个前台进程。前台进程会阻塞Shell。如果Shell启动前台进程，那么在Shell中的输入就重新定向到前台进程的标准输入。Shell的按键信号，自然也是发送给前台进程的。

值得注意的是，Shell除了可以有一个前台进程，还可以有多个后台进程。后台进程不会阻塞Shell的命令行，比如：

```
$ping localhost > log &
```

按键信号不会发送给后台进程。

Linux使用特定的短语来指代一种信号。可以用按键发出的信号有三个，这三个信号都是传递给Shell的前台进程的。

- SIGINT：按快捷键Ctrl+C，**中断**（Interrupt）前台进程。
- SIGTSTP：按快捷键Ctrl+Z，**暂停**（Stop）前台进程。
- SIGQUIT：按快捷键Ctrl+\，**退出**（Quit）前台进程。

23.2 kill命令

按键是向前台进程发出信号的快捷方式，我们也可以用kill命令向特定进程发出信号。首先在一个Shell中运行一个前台进程：

```
$ping localhost > log
```

在另一个Shell中，先找到ping命令对应进程的PID：

```
$ps aux | grep "ping localhost"
```

找到进程的PID，如8577，就可以用kill命令中断程序：

```
$kill 8577
```

事实上，kill命令可以向系统中的任何一个进程发信号，无论这个进程是前台进程还是后台进程，以一个后台进程为例：

```
$ping localhost > log &
```

后台进程会在Shell上打印出PID，比如9949。有了进程的PID，我们可以用kill向进程发信号：

```
$kill -s SIGTSTP 9949
```

命令kill能发出信号的种类比快捷按键多很多，比如下面的两个信号。

- SIGCONT：通知暂停的进程继续。

- SIGALRM：定时信号。

可以用下面的命令来查询完整的信号列表：

```
$kill -l
```

还可以用man命令来查询更详细的文档：

```
$man 7 signal
```

我们依然用kill发出这些信号，比如让暂停的进程继续：

```
$kill -s SIGCONT 9949
```

23.3 信号机制

信号是Linux系统的重要机制，我们有必要理解它的原理。信号本质上很简单，就是内核传递给进程的一个整数。每个整数代表了一种信号，如表23-1所示。

表23-1 整数与信号

整 数	信 号
2	SIGINT
3	SIGQUIT
14	SIGALRM

可以用下面的命令来查询信号对应的整数：

```
$kill -l
```

我们可以把信号想象成大楼管理员往住户的信箱里塞的小纸条，如图23-1所示。所谓的大楼管理员就是Linux内核，而住户就是进程。在内核的内存空间里，给每个进程预留了一块用于存放信号的空间。这个空间就是内核和住户之间沟通的信箱，信箱里能存放的内容就是对应了某种信号的整数。



图23-1 信号：信箱里的小纸条

需要发信号的情况有很多。它可以是内核自身产生的，比如出现分母为0的除法运算这样的错误，内核就会用SIGFPE信号来通知进行该运算的进程。信号也可以是其他进程产生的，比如用户用快捷键发出的中断信号，实际上是由Shell产生的。无论是哪种情况，信号最终都是由内核写入目标进程的信箱。这样，就生成了信号。

信号生成后，就会一直躺在信箱里，直到住户来查看。信号总是在特定的时机下被查看。每次进程完成系统调用、即将退出内核的时间，就是查看信号的最好时机。原因很简单，信号保存在内核空间，而进程执行系统调用时正好处于内核模式，查看内核空间的代价最小。如果有信号，那么进程会执行对应该信号的操作，这称作**信号处理**（Signal Disposition）。从信号生成到信号处理这段时间，信号处于**等待**（Pending）状态。

可见，信号只是一个整数，信息量很小。信号的运行机制也很简单，就像是一个投递到信箱里的小纸条。正因如此，信号便于管理和使用，因此信号总是那些涉及系统运行的关键任务，比如通知进程终结、中止或者恢复等。

23.4 信号处理

我们之前用信号来让进程中断或恢复。事实上，信号并没有操纵进程的魔力。它只能通知进程某种情况的发生。在刚才ping命令的例子中，ping收到信号后，根据Linux系统的管理，针对每种信号都采取了对应的默认操作，但这并不绝对。进程的信号处理，有下面三种可能。

- **无视信号**（Ignore Signal）：信号被清除，进程本身不采取任何特殊的操作。
- **默认操作**（Default Action）：每个信号对应有一定的默认操作，比如SIGCONT用于继续进程。
- **捕获信号**（Catch Signal）：根据信号，执行程序中自定义的操作。

收到信号后，进程会根据信号的种类和程序设计，决定采取哪种行动。特别是在捕获信号的情况下，进程收到信号后，往往会进行一些长而复杂的操作。

我们先以bash脚本为例，说明信号处理。bash脚本运行后成为一个进程，这个进程就可以处理信号。先来写一个bash脚本，名为test_signal.bash：

```
#!/bin/bash

while true
do
    echo hello
done
```

这个脚本不断地打印出文本“hello”。运行脚本：

```
$/test_signal.bash > log &
```

获得脚本的PID，例如10412。我们可以向脚本进程发出信号，比如：

```
$kill -s SIGINT 10412
```


此时进程对信号的处理，采用的是默认操作。我们可以利用trap命令，让一个bash脚本来捕获信号：

```
#!/bin/bash

trap "echo 'interrupted'; exit 1;" SIGINT

while true
do
    echo hello
done
```

命令trap后面跟的参数，一个是捕获信号后想要进行的操作，一个是用来说明想要捕捉信号的名称。除了用信号名，你也可以用信号编号。根据这个脚本中的trap命令可知，这个脚本会捕获SIGINT信号，并针对该信号进行自定义的处理：先打印“interrupted”，再以状态1退出。

我们可以配合bash的函数，在捕获信号后进行更复杂的操作。

```
#!/bin/bash

WANNA_QUIT=false

function handle_exit {
    if [ "$WANNA_QUIT" = true ]; then
        echo "Bye."
        exit 0
    else
        WANNA_QUIT=true
        echo "Press Ctrl+C again to quit."
    fi
}

trap "handle_exit;" SIGINT

while true
do
    echo working...
    sleep 1

done
```

在这个脚本中我们定义了一个叫作handle_exit的函数，这个函数在第一次被调用时不会直接退出，而是提示用户如果退出，需要再次按下快捷键Ctrl+C，即打印提示“Press Ctrl+C again to quit.”。而第二次收到SIGINT信号后，将会输出Bye，并退出程序。

注意，脚本一开始就用trap说明了信号捕获的相关操作，但脚本不会停在trap那一行等待信号。脚本继续执行下去，进行循环操作。只有等到信号出现时，脚本才会回到trap说明的操作。因此，对于trap命令来说，语句执行的时机和语句发生作用的时机是分离的。回忆前面介绍的echo、ls、which等语句，当语句执行时，语句规定的操作就进行了，这种工作方式是**同步**（Synchronous）。而trap的工作方式是**异步**（Asynchronous）的，异步编程用于处理像信号这样出现时机不确定的事件。

23.5 C程序中的信号

不仅可以在bash脚本中捕获信号，也可以在C语言编写的程序中捕获信号。C语言信号相关的内容在头文件 *signal.h* 里。捕获信号的语句格式如下：

```
signal(SIGINT, sigint_handler);
```

signal函数跟着两个参数，第一个是信号标识，在这里捕捉的是SIGINT，第二个是回调函数，这里是sigint_handler。SIGINT常量是在*signal.h*中定义的，如果打开头文件，那么可以看到其中有下面这一行：

```
#define SIGINT 2 /* interrupt */
```

回调函数和一般的C语言函数类似，例如定义下面这个函数：

```
void sigint_handler()
{
    printf("Received SIGINT.");
}
```

这个C语言程序运行时，如果遇到SIGINT信号，就会打印“Received SIGINT.”这样一句话。

我们来看一个完整的C程序：

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void sigint_handler()
{
    printf("Received SIGINT\n");
}

int main()
{
    signal(SIGINT, sigint_handler);

    while (1)
    {
        printf("waiting...\n");
        sleep(1);
    }

    return 0;
}
```

上面的程序包含了一个无限循环，程序在这个循环中反复打印一串文本。编译执行后，运行效果如下：

```
waiting...
waiting...
waiting...
```

程序不断打印“waiting...”。此时，发出信号，即按下快捷键Ctrl+C，将触发程序一开始定义的信号处理函数sigint_handler。程序将执行该处理函数中的语句，即打印：

```
Received SIGINT
```

第4部分 深入Linux

本部分将会介绍Linux的高级概念，以及内核的主要功能。这一部分通常会出现在“Linux高级编程”或“操作系统原理”这类计算机高级课程中。笔者把这部分的知识框架从庞杂的C代码中抽取出来，想让每位读者都能以“玩”的心态来欣赏Linux底层的设计。与前面的部分相比，这一部分更加注重抽象概念。当然，你也可以先跳到第5部分实践，在熟悉了树莓派和Linux系统后，再来阅读本部分。

第24章 进程的生与死

操作系统把计算机活动划分成进程。程序员编写的程序，也必须运行成进程，才能出现实际效果。既然进程在计算机活动中拥有如此关键的地位，那么我们理应更深入地了解进程。本章将介绍进程的创建和终结，以及与之相关的进程权限。

24.1 从init到进程树

计算机开机时，Linux内核只创建了一个名为init的进程。在Linux运行期间，会有很多其他新进程，如Shell进程、音乐播放程序进程、邮件程序进程等。Linux内核不直接创建其他新进程，除了init进程之外的所有进程，都是通过fork机制创建的。

所谓的fork，就是从老进程中复制出一个新进程，英文中的“fork”是“分叉”的意思，如图24-1所示。老进程就像一条带有分叉的小溪。老进程分出新进程后，老进程继续运行，成为新进程的**父进程**（Parent Process），新进程成为老进程的**子进程**（Child Process）。

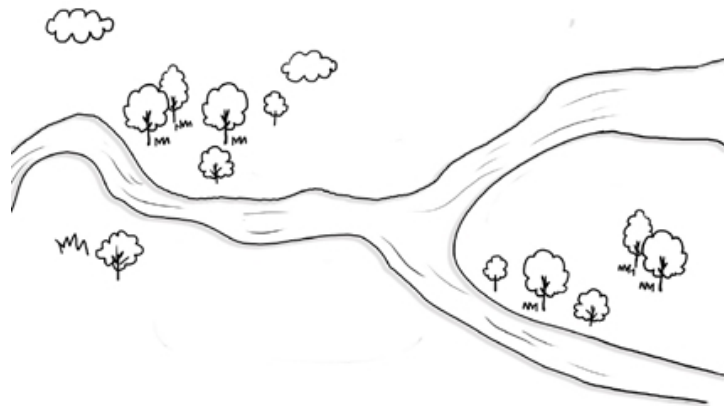


图24-1 fork：分叉

查询当前Shell下的进程：

```
$ps -o pid,ppid,cmd
```

结果如下：

```
PID  PPID  CMD
16935  3101  sudo -i
16939  16935  -bash
23774  16939  ps -o pid,ppid,cmd
```

可以看到，第二个进程bash是第一个进程sudo的子进程，而第三个进程ps是第二个进程的子进程。

一个进程除了有一个PID之外，还会有一个PPID（Parent PID），即用来存储的父进程PID。子进程可以通过查询自己的PPID来了解自己的父进程。从任何一个进程出发，循着PPID不断向上追溯，总会发现源头是init进程，因此Linux的所有进程也构成了一个树状结构，这个树状结构以init进程为根。我们可以用pstree命令来显示树莓派的整个进程树。

在Linux中，fork无处不在。就拿Shell来说，当我们执行一个命令时，Shell进程就会fork一个子进程，用于执行命令对应的程序。在编写应用程序的过程中，也会用到fork机制，从而让一个新的进程来执行子任务。

24.2 fork系统调用

Linux的应用程序可以通过fork系统调用来创建新进程。该系统调用发生后，就有父与子两个进程，而且两者的进程空间完全相同。

这就创造了一个“我是谁”的问题，即其中的一个进程如何知道，自己是父进程，还是子进程。Linux内核已经考虑到了这一点，并通过fork调用的返回值解决了问题。由于fork之后有两个进程，fork系统调用会返回两次。一次返回到父进程，把子进程的PID作为返回值交给父进程。如果fork不成功，那么fork调用就会返回一个负值给父进程。

另一次返回到子进程，用0作为返回值。通过检查fork调用的返回值是否为0，进程就可以知道自己是否是子进程。一方面，父进程可以通过fork的返回值知道子进程的PID。另一方面，进程又可以通过自己的PPID来知道父进程是谁，这样，进程就能明白自己在进程树中的位置了。

我们来看一个fork的例子。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t pid;

    int var = 1024;

    pid = fork();

    if (pid < 0) {
        printf("fork error");
        exit(1);
    } else if (pid == 0) {
        /* 子进程 */
        printf("child: %d\n", var);
    } else {
        /* 父进程 */
        sleep(1);
        printf("parent: %d\n", var);
    }

    return 0;
}
```

程序中的getpid函数用于获得当前进程的PID。当fork返回后，内存中有两个进程。通过if区分出父进程和子进程，父进程将打印：

```
parent: 14813
```

子进程将打印：

```
child: 14814
```

可见，进程通过fork返回弄清了自己是子进程还是父进程，就能根据自己的情况执行不同的任务。进程在获知自己是子进程后，通常会通过exec系列函数中的一个来加载新的程序文件，从而与父进程执行不同的任务。比如Shell执行ls命令，会先fork自己的进程，然后在子进程中运行 */bin/ls* 这一程序文件。

24.3 资源的fork

进程空间记录进程的数据和状态。当进程fork时，Linux需要在内存中分配新的进程空间给新进程。此外，进程空间中的内容记录了进程的状态和数据。因此，原有进程空间中的所有内容，如程序段、全局数据、栈和堆，都要复制到新的进程空间中。在下面的程序中，子进程和父进程在fork之后有相同的栈，自然也会有相同的变量var。


```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t pid;

    int var = 1024;

    pid = fork();

    if (pid < 0) {
        printf("fork error");
        exit(1);
    } else if (pid == 0) {
        /* 子进程 */
        printf("child: %d\n", var);
    } else {
        /* 父进程 */
        sleep(1);
        printf("parent: %d\n", var);
    }

    return 0;
}

```

除了进程空间，fork还会复制**进程描述符**（Process Descriptor）。内核中保存了每个进程的相关信息，即进程描述符。描述符是进程在内核中的“驻联合国代表”。每一个进程都会在内存中有一个对应的进程描述符。之前提到的PID、PPID和信号，都保存在进程描述符中。

在fork之后，系统出现了一个新的进程，内核就要增加对应该进程的描述符。子进程描述符中的很多内容都是从父进程的描述符中复制过来的。

- 当前工作目录。
- 环境变量。
- 已打开文件的相关信息。

- 信号mask和disposition。

这些信息都是程序运行必需的信息。如果子进程和父进程继续执行同一个程序，那么上述附加信息的改变，会造成子进程运行的错误。比如，父进程打开了一个文件，那么fork出的子进程也可以正常打开文件。

父进程和子进程描述符有很多信息不同。

- PID、PPID。
- 进程运行时间的相关信息在子进程中重置为0。
- 父进程的文件锁在子进程中被清空。
- 父进程的未处理信号在子进程中被清空。

这些信息都是描述进程个体特征的信息。子进程中描述符如果复制上述信息会出问题。我们已经知道，子进程和父进程的PID、PPID必然不同。此外，如果子进程的运行时间不重置为0，那么子进程运行时间的统计就不正确。可见，进程描述符是否复制某一块信息，最重要的原则是保证新进程的正确运行。

24.4 最小权限原则

Linux有一个“**最小权限**”（Least Privilege）的原则，就是收缩进程所享有的权限，以防进程滥用特权。进程权限也是根据用户身份进行分配的。然而，进程的不同阶段可能需要不同的特权。比如运行到中间时，需要先以更高的权限读入某些配置文件，再进行低权限的处理操作。由于权限和用户身份挂钩，这意味着进程需要在不同身份之间变化。

用户启动进程会让这个进程有3个身份：真实身份、存储身份和有效身份。每个身份都包含一套UID和GID。其中，真实身份是用户登录使用的身份。存储身份如果设置，就是程序文件的拥有者。有效身份则是判断进程权限时使用的身份。

在进程的运行过程中，进程可以从真实身份和存储身份中选择一个，复制到有效身份。通过这种机制，进程就可以在运行过程中变换权限。如果操作所需权限同时超越了真实身份和存储身份的权限，那么无论如何变换身份进程都无权操作。此外，并不是所有的程序都需要设置存储身份。需要这么做的程序文件会把权限的执行位上的“x”改为“s”。这时，用户权限的这位位叫作**设置 UID 位**（Set UID Bit），而组权限的这位位叫作**设置 GID 位**（Set GID Bit）。

24.5 进程的终结

进程总有终结的时候。进程可以自发终结，比如进程在main函数结尾调用return，或者在程序中的某个位置调用exit函数直接退出。我们在信号中也看到，进程可以根据信号终结。此外，当进程出现致命错误时，比如当进程出现栈溢出错误时，内核也会主动终结进程。

进程终结时，会有一个退出码。这个退出码可以是return或exit返回的，也可以是内核强制终结进程时设置的。当程序正常退出时，程序的退出码为0。如果运行过程中有错误或异常状况，那么退出码会是大于0的整数。退出码可以代表进程退出的原因。

当某个进程终结时，父进程会获得通知，进程空间随即被清空，然而，进程附加信息会保留在内核空间中。也就是说，即使一个进程终结了，它还是会在内核中留下痕迹。删除进程对应内核信息的重任，就落在父进程身上。

按照Linux的惯例，父进程有义务对子进程使用wait系统调用。在调用wait之后，父进程暂停，等待子进程终结。子进程终结后，父进程能从内核中取出子进程的退出信息，并清空这个子进程的进程描述符。在完成了上述工作之后，父进程将恢复运行。下面是一段wait程序的示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{

    pid_t pid = fork();
    if (pid < 0) //无法创建子进程
    {
        printf("出错啦! ");
    }
    else if (pid == 0) //子进程
    {
        //做一些很漫长的运算
        int sum = 0;
        for (int i = 0; i < 100000000; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                sum += i;
            }
        }
        exit(0);
    }
    else //父进程
    {
```

```

    int status;
    printf("子进程 PID %d...\n", pid);
    //等待子进程结束
    do
    {
        waitpid(pid, &status, WUNTRACED);
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));

    //status 是子进程返回值
    printf("子进程返回值 %d\n", status);
}
}

```

如果父进程早于子进程终结，子进程就会和进程树失联，成为一个**孤儿进程**（Orphan Process）。孤儿进程会过继给init进程，因此进程init是所有孤儿进程的父进程。而对孤儿进程调用wait的责任，也就转交给了init进程。

当然，wait系统调用只是约定俗成的责任。Linux对此没有强制规定，一个程序也完全可以不对子进程调用wait，但这种程序会导致子进程的退出信息滞留在内核中。在这样的情况下，子进程成为**僵尸进程**（Zombie Process）。当大量僵尸进程积累时，内核空间会被挤占，优秀的程序员应该杜绝这种情况的发生。

第25章 进程间的悄悄话

有了进程空间的概念，我们可以看到进程的独立性。每个进程的数据停留在自己的进程空间里，互不干涉。这样的独立性，让每个进程可以专注于自己的任务，大大减少了进程间相互干扰而出错的可能性。然而，有的时候，我们又需要打破这种独立性，让进程之间分享数据，从而协调工作。这个时候，就需要进行**进程间通信**（IPC，Inter-process Communication）。

25.1 管道

从广义上说，任何能在进程间传送信息的方式都属于IPC。我们先来回顾一些已经接触过的IPC的方式。一种原始的IPC方式就是进程间通过文件来交换信息，即一个进程往文件里写数据，另一个进程从文件中读出数据。

```
$ping localhost > log.txt &  
$while true; do tail -1 log.txt; done;
```

在上面的命令中，ping的进程持续运行，并向文件 *log.txt* 中输出。随后，我们循环启动tail进程。每次这个进程都从 *log.txt* 读出最后一行。这种形式的数据交换发生在 *log.txt* 存活的SD卡中，而不是ping或tail进程空间所在的内存，因此可以绕开进程空间相互独立的限制。但相对于内存，存储器读写速度慢，所以这个方式效率低。此外，多进程同时写入同一个文件，很容易造成文本混乱。

信号也算是一种IPC。一个进程发出信号，放入目标进程的描述符。另一个进程进行系统调用时，在自己的描述符中看到该信号。两个进程通过信号进行了简单的数据交换。信号的一大缺点是无法大量交换数据。信号的数据交换发生在内核的进程描述符中，因为内核空间和进程空间独立，所以也绕开了进程空间相互独立的限制。

在介绍Linux文本流时，我们使用了管道。管道直接把一个进程的输出和另一个进程的输入连接起来。由于管道实现了进程间的数据交换，因此管道也是一种IPC。通过管道，进程间可以通过文本传输大量数据。我们已经在Shell中实验过管道。

```
$ping localhost | cut -c 1-24
```

这个命令会同时启动两个进程，分别运行ping和cut的程序。每当ping输出到标准输出时，输出信息会通过管道传递给cut的进程。前者负责探测网络，后者负责裁剪输出，各司其职，又可以协同合作。由于管道兼具速度和数据量优势，因此在IPC方面，管道要比文件和信号都常用。

基于管道的进程间数据交换也发生在内核空间。与信号类似，它也是通过内核空间来绕开进程空间的独立性限制。创建管道之后，这个管道在内核空间里会有一个专用的缓冲区，用于存放要传输的数据。输出进程会向这个缓冲区不断地写入数据，而输入进程会从缓冲区按照顺序取出数据。从效果上来看，数据沿着管子有序地从一个进程流入另一个进程。

管道的缓冲区不需要很大。它被设计成环形的数据结构，可以循环利用。当旧的数据已经取出时，新数据就可以占用这块内存空间了。如果缓冲区中没有数据，则从管道中读取的进程会等待，直到另一端的进程放入数据。如果缓存区放满数据，则尝试放入数据的进程会等待，直到另一端的进程取出信息。当两个进程都终结时，管道会自动消失，缓冲区的内存空间也会收回。

25.2 管道的创建

Linux创建管道的经典方式，是利用fork机制。管道基于文本流，打开和操作方式类似于Linux中的文件。在介绍资源的fork时提到过，进程会把打开文件的信息复制给子进程。这样进程输入和输出端口的信息也会复制到子进程。因此，一个进程会像新建文件一样，先创建一个管道，该进程的输入和输出，都直接接在这个管道上。

完成了上述准备步骤之后，进程会fork。随着资源的fork，进程到管道的两个连接也复制到了子进程上，两个进程同时接上同一个管道。随

后，每个进程关闭自己不需要的一个连接。父进程关闭从管道来的输入连接，即子进程输出到管道的连接。这样，剩下的连接就形成了可以从一个进程向另一个进程传输的管道。

由于这种创建管道的方式基于fork机制，因此管道必须用于父进程和子进程之间，或者拥有相同祖先的两个子进程之间。为了解决这一问题，Linux提供了**命名管道**（Named Pipe）。

命名管道的基础是FIFO（First In First Out）。FIFO是一种特殊的文件类型，它在文件系统中具有对应的路径。如果一个进程以读的方式打开FIFO文件，而另一个进程以写的方式打开同一文件，那么内核就会在这两个进程之间建立管道。虽然用法上和文件一样，但FIFO存活于内核空间，所以它的IPC效率比存储器文件的IPC高得多。

FIFO文件的命名，其实就是管道遵循的“先进先出”的队列数据结构，从而保证信息的有序传输。虽然命名管道和无名管道的运作方式相同，但FIFO借用了文件系统，在文件系统中表征成一个文件。只要两个进程读写同一FIFO文件，就可以自由地建立管道，可以直接用命令来创建命名管道：

```
$mknod {文件名} p
```

例如，下面的命令可以在临时文件夹创建一个命名管道。

```
$mknod /tmp/named-pipe p
```

打开两个命令行窗口，在窗口1中输入：

```
$tail -f /tmp/named-pipe
```

在窗口2中输入：

```
$echo hello >> /tmp/named-pipe
```

返回窗口1，我们就会看到echo的内容被命名管道传递到了这边。

使用rm命令删除FIFO文件时，命名管道连接也随之消失。由于可以借用文件系统的创建、搜索和删除功能，命名管道在管理上更加方便。

管道是Linux系统下非常常用的IPC方式。在Linux系统中，管道共用了存储器文件的API，所以从创建到使用都很方便，但管道有时也不能完全满足我们进行IPC的需要。下一节将介绍管道之外的其他IPC方式。

25.3 其他IPC方式

本节介绍的IPC都有悠久的历史。它们通过不同的方式，实现了进程间的资源共享。我们分别来看这些IPC方式。

1. 消息队列

消息队列（Message Queue）与管道有些类似。它也是在内核空间中把数据排好队，先放入队列的消息被最先取出。正如名字就已经说明的，消息队列的数据单元是一条长度不定的消息，这一点和管道以字节为单位的数据流不同。此外，管道两端都只能连接一个进程。而消息队列允许多个进程参与，既可以有多个进程往队列中放入消息，也可以有多个进程从队列中取出消息。因为消息队列不依附于特定的进程，所以消息队列不会像管道那样自动消失。消息队列一旦创建，会一直留在内核空间中，直到某个进程删除该队列。像流水线一样的消息队列，如图25-1所示。

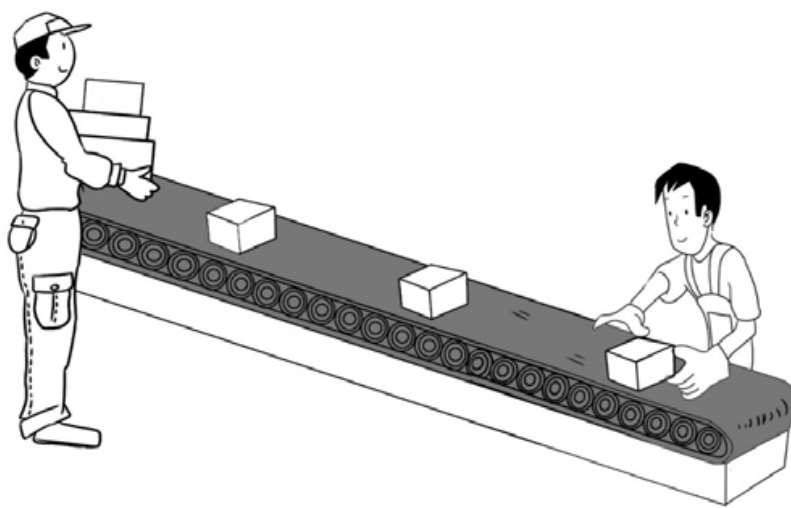


图25-1 像流水线一样的消息队列

消息进入队列就已经排好序了。某个进程从队列中取出消息时，按照先进先出的顺序逐个取出所有消息。消息放入消息队列时，还可以带

有一个整数，作为该消息的类型。当进程取出消息时，也可以提供类型参数，按照先进先出的顺序，只取出某个类型的消息。在这种时候，类型就起到了筛选消息的功能。

需要注意的是，在使用消息队列时，要注意及时删除队列，以免造成内核空间的浪费。

下面，我们用C语言编写使用消息队列传输数据的程序。

发送端：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_struct {
    int type;
    char content[100];
} message;

int main()
{
    //生成 IPC Key
```

```

key_t key = ftok("path", 65);

//创建一个 Message Queue, 获得 Message Queue ID
int mqid = msgget(key, 0666 | IPC_CREAT);

//输入文本
printf("请输入要发送的文本: ");
fgets(message.content, 100, stdin);
message.type = 1;

//发送数据
msgsnd(mqid, &message, sizeof(message), 0);
printf("发送的数据: %s", message.content);

return 0;
}

```

接收端:

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_struct {
    int type;
    char content[100];
} message;

int main()
{
    //生成 IPC Key
    key_t key = ftok("path", 65);

    //创建一个 Message Queue, 获得 Message Queue ID
    int mqid = msgget(key, 0666 | IPC_CREAT);

    //收取数据
    msgrcv(mqid, &message, sizeof(message), 1, 0);
    printf("收到的数据: %s", message.content);

    //销毁 Message Queue
    msgctl(mqid, IPC_RMID, NULL);

    return 0;
}

```

发送方运行效果如下所示。

```
$/sender  
请输入要发送的文本：你好，世界。  
发送的数据：你好，世界。
```

接收方的运行效果如下所示。

```
$/receiver  
收到的数据：你好，世界。
```

2.共享内存

共享内存（Shared Memory）的IPC方式从另一个思路出发，直接打破了进程空间的独立性限制。一个进程可以将自己内存空间中的一部分拿出来作为共享内存，并允许其他进程直接读写。在这个过程中，数据始终停留在同一个地方，不需要迁移到存储器空间或内核空间，所以它是效率最高的IPC方式。

共享内存特别适用于大数据量的情景，比如图像处理。图像处理时可能用四个进程分别负责图像读取、图像旋转、图像平滑和图像存储。这样一种流水线式的多进程协同方式，特别适用于多核的CPU。如果按照之前的IPC方式，大尺寸的图像数据在进程间接力时必须复制到内核空间或存储器，则会大大降低计算机的处理速度。这个时候，共享内存就是最好的IPC方式。

图像读入进程把存储器中的图像数据放入自己的内存空间，并把该部分内存空间设置为共享内存。此后的图像选择、图像平衡和图像存储进程，都会直接读写第一个进程的内存区域。从图像数据进入内存开始，直到处理后的数据存储，图像数据都待在内存中的同一个位置，计算机的处理效率自然大为提高。共享内存就像是两个人耕耘同一片田地，如图25-2所示。



图25-2 共享内存：两个人耕耘同一片田地

一个用共享内存在父子进程间共享数据的C语言例子如下。

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>

#define ANSI_COLOR_CYAN    "\x1b[36m"
#define ANSI_COLOR_RESET  "\x1b[0m"

void *create_shared_memory(size_t size)
{
    //将共享内存设置成可读可写
    int protection = PROT_READ | PROT_WRITE;

    //将共享内存设置成为共享（第三方进程可读）和匿名（第三方进程无法得到访问地址）
    int visibility = MAP_ANONYMOUS | MAP_SHARED;

    //创建共享内存
    return mmap(NULL, size, protection, visibility, 0, 0);
}

int main()
{
    setbuf(stdout, NULL);

    void *shmem = create_shared_memory(128);

    int pid = fork();

    if (pid == 0)
    {
        while (1)
        {
            char message[100];
            printf("请输入要发送的文本: ");
            fgets(message, 100, stdin);
            memcpy(shmem, message, sizeof(message));
            printf("子进程成功写入数据: %s\n", shmem);
        }
    }
    else
    {
        while (1)
        {

```

```

        printf(ANSI_COLOR_CYAN "父进程内存中的数据: %s\n" ANSI_COLOR_RESET,
shmem);
        sleep(1);
    }
}
}

```

运行程序后，父进程的内容将会用蓝色输出，子进程的内容会用默认颜色输出。用键盘输入一段文字将会被子进程捕获后修改共享内存，接着父进程将会获得修改过的内容。

3.套接字

套接字（socket）也是一种常见的IPC方式，在互联网通信上扮演着关键角色。我们可以把互联网通信也理解成IPC问题，只不过这个时候的多个进程可以分布在不同的电脑上。互联网上常用的网络协议都可以通过套接字的方式连接，从而连接分别位于两台计算机上的两个进程。比如，可以把访问某个网站看作是本地电脑的浏览器进程和远程电脑的服务器进程的一次套接字IPC。通过这样一次IPC，本地电脑和远程电脑交换了数据，我们也就看到了本来不存在于我们电脑上的网页内容。网络通信超出了本书的范围，这里就不过多深入网络套接字了。

第26章 多任务与同步

上一章提到了IPC，实际上它涉及一个关键问题：计算机的并发性。Linux系统是一个支持**并发**（Concurrency）的操作系统。并发系统可以同时执行多个任务。多个进程通过IPC的数据沟通，可以合作完成一个复杂任务。然而，并发系统并不简单，必须解决同步的问题。

26.1 并发与分时

在过去很长时间里，计算机使用的都是单核CPU。每个时刻，单核的CPU只能执行一条指令。从指令的角度看，单核CPU计算机不能并发。

但单核CPU计算机可以同时运行多个任务。这种并发是通过**分时**（Time-Sharing）来实现的。所谓的“分时”，就是把时间分配给多个服务对象。这就好像一位妈妈同时照顾三个婴儿。她先给第一个孩子端上饭，然后给第二个孩子倒水。倒完水之后，她又跑去给第三个孩子梳头。妈妈把自己的时间分给了三个孩子。虽然每个特定的时刻，妈妈只能照顾一个孩子，但三个孩子还是能一直感受到妈妈的温暖。照顾多个进程的CPU类似于照顾三个孩子的母亲，如图26-1所示。



图26-1 照顾多个进程的CPU类似于照顾三个孩子的母亲

和母亲照顾多个孩子的情况类似，CPU的工作时间也可以分配给多个进程。CPU执行进程A一段时间，就换进程B继续执行。切换进程的工作由操作系统负责，操作系统会先把进程A的状态更新到进程A的描述符中，再根据进程B描述符中的记录，从进程B上次暂停的地方继续进行下去。这样，多进程协作的目的就可以实现。即使在单核计算机上，我们也可以边浏览网页边听音乐，也就是说，单核CPU也可以让多个任务同时推进。

现代的多核CPU，可以同时执行多个指令，从基础的物理层面实现了并发。也就是说，现代的计算机看起来像是多位保姆照顾多个婴儿。即便如此，操作系统还是会用分时的方式来安排任务。原因很简单，计算机中的任务总数很容易超过CPU可以同时执行的指令总数。此外，通过分时系统，计算机的运行效率也能有效提升。我们将在讲解调度器的时候，深入这一点。

26.2 多线程

第26.1节中的并发是通过多个进程实现的。多进程加上IPC，就已经提供了丰富的多任务协作方式。如果调出其中的一个进程看，它的内部只进行一个任务，不会有并发。进程就好像一位专心写作业的小朋友，不会同时看电视。这种注意力单一、每个时刻只做一件事的工作方式，叫作**单线程**（Single-Threading）。我们前面见过的进程，都是单线程进程。

但程序员很多时候会在一个进程内部运行**多线程**（Multi-Threading）。多线程允许在一个进程中同时执行多个子任务。由于我们要同时关照多个线程的状态，进程的结构必须发生变化。

- 进程描述符需要记录每个线程的相关信息，特别是它们的状态和进度。这一点和进程的情况类似。
- 操作系统需要把适当的计算时间分配给进程。内核调度器在分配计算时间时，必须把各个线程考虑在内。
- 进程空间中必须有多个栈。

前两者和进程的情况类似。着重看最后一点，即进程空间中必须有多个栈。栈记录着函数调用的顺序，最下方的帧是唯一一个激活函数。

既然多线程是多任务并发，那就意味着会有多个函数处于激活状态，并同时运行。比如下面的多线程程序：

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> // for sleep
```

```

void *func1(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        printf("func1 is running %d \n", i);
        sleep(1);
    }
    return NULL;
}

void *func2(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        printf("func2 is running %d \n", i);
        sleep(1);
    }
    return NULL;
}

void *func3(void)
{
    int i;
    for (i = 0; i < 3; i++)
    {
        printf("func3 is running %d \n", i);
        sleep(1);
    }
    return NULL;
}

int main()
{
    int i = 0, ret = 0;
    pthread_t func1_id, func2_id, func3_id;

    ret = pthread_create(&func1_id, NULL, (void *)func1, NULL);
    if (ret)
    {
        printf("Cannot create func1.\n");
        return 1;
    }

```

```

ret = pthread_create(&func2_id, NULL, (void *)func2, NULL);
if (ret)
{
    printf("Cannot create func1.\n");
    return 1;
}

ret = pthread_create(&func3_id, NULL, (void *)func3, NULL);
if (ret)
{
    printf("Cannot create func1.\n");
    return 1;
}

// Wait for func3.
pthread_join(func3_id, NULL);

printf("Main thread exists.\n");

return 0;
}

```

在C语言中，可以用pthread的一系列函数来操作多线程。我们把某个函数放入到新线程中执行，如func1()，程序输出如下所示。

```

func1 is running 0
func2 is running 0
func3 is running 0
func1 is running 1
func2 is running 1
func3 is running 1
func1 is running 2
func2 is running 2
func3 is running 2
func2 is running 3
func1 is running 3
Main thread exists.

```

程序的运行流程是一个多线程的流程，如图26-2所示。

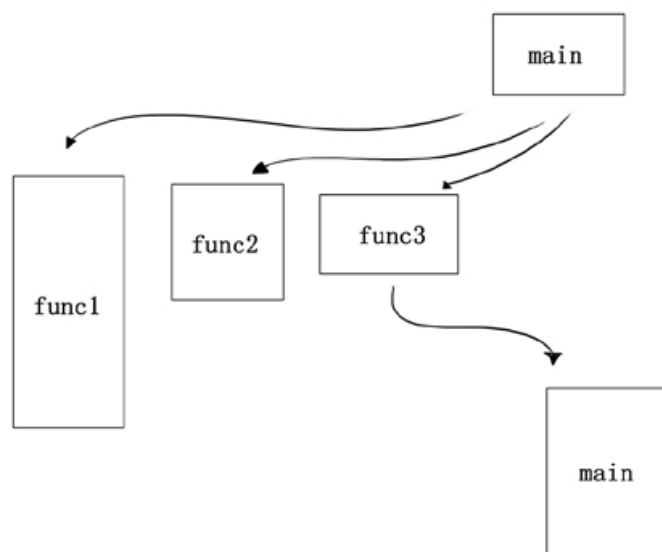


图26-2 多线程的流程

从ain()到func3()再到main()构成一个线程，而func1()和func2()构成另外两个线程。

当程序创建一个新的线程时，必须为这个线程建一个新的栈，每个栈对应一个线程。当某个栈执行到全部弹出时，对应线程完成任务。因此，多线程的进程在内存中有多个栈。多个栈之间以一定的空白区域隔开，以备栈的增长。对于多线程来说，由于同一个进程空间中存在多个栈，任何一个空白区域被填满都会导致栈溢出的问题。

进程空间上需要调整栈的部分。一个线程与其他线程共享内存中的程序段、堆和全局数据。这些部分的组织方式也和单线程进程类似。由于多线程共享了很多内存区域，它们都可以直接读写堆上的内容，线程间的数据共享变得很简单。因此，多线程的数据交流成本要比多进程低得多。这也是程序员使用多线程的一大原因。

26.3 竞态条件

多进程和多线程都实现了并发。并发系统实现了多任务协作，但容易产生**竞态条件**（Race Condition）。如果多个任务可以共享数据，特别是可以同时修改某个数据时，就很有可能发生竞态条件。

我们来看竞态条件在现实生活中的例子。如果一节火车有100张票，同时在10个售票窗口销售。每位售票员卖完一张票之后，就打电话告诉总部卖出去一张票，可售卖的票数就会减1。

用一个多线程程序来重现上述情形。程序用全局变量*i*存储剩余的票数。多个线程不断地卖票，也就是从*i*中减去1，直到剩余票数为0，因此每个都需要执行如下操作^[1]：

```
/*mu 是一个全局互斥锁 */

while (1) {                                /*无限循环*/
    if (i != 0) i = i - 1
    else {
        printf("no more tickets");
        exit();
    }
}
```

每个线程会进行两件事。一件事是判断是否有剩余的票，即判断*i*是否等于0。另一件事是卖票，即从*i*上减去1。这两件事情之间存在一个时间窗口，其他线程可能在此时间窗口内执行卖票操作，即从*i*中减1。但之前的卖票线程已经执行过了判断，不知道*i*发生了变化，所以会继续执行卖票，以至于卖出不存在的票，让*i*成为负数。对于一个真实的售票系统来说，这将成为一个严重的错误。

在并发情况下，指令执行的先后顺序由内核决定。同一个线程内部，指令按照先后顺序执行，但不同线程之间的指令很难说清哪一个会先执行。这个时候，如果并发任务可以同时读取同一块数据，就会造成结果难以预测的情况。因此，在并发系统中，如果运行的结果依赖于不同线程执行的先后顺序，则会造成竞态条件。

26.4 多线程同步

对于并发程序来说，**同步**（Synchronization）是指在一定的时间内只允许某一个任务访问某个资源。同步可以解决竞态条件的问题。比如，某段时间内只能有一个售票员查询票数并售出，其他售票员在此期间不能售票，就不会有竞态条件的问题。

以多线程为例，多线程同步就是在一定的时间内只允许某一个线程访问某个资源。在多线程中，我们可以通过**互斥锁**（Mutex）、**条件变量**（Condition Variable）和**读写锁**（Reader-Writer Lock）来同步资源，分别来看它们的功能。

1.互斥锁

互斥锁是一个特殊的变量，它有锁上和打开两个状态。互斥锁一般被设置成全局变量。打开的互斥锁可以由某个线程获得。一旦获得，这个互斥锁会锁上，此后只有该线程有权打开。其他想要获得互斥锁的线程，要等到互斥锁再次打开的时候。

我们可以将互斥锁想象成为只能容纳一个人的洗手间，当某个人进入洗手间时，可以从里面将洗手间锁上。其他人只能在互斥锁外面等那个人出来，才能进去。在外面等候的人并没有排队，谁先看到洗手间空了，就可以先冲进去。上面的问题很容易使用互斥锁模拟，每个线程的程序可以改为：

```
/*变量 mu 是一个全局的互斥锁*/

while (1) {                                /*无限循环*/
    mutex_lock(mu);                        /*获得互斥锁并锁上。如果不能获得，就等待*/
    if (i != 0) i = i - 1;
    else {
        printf("no more tickets");
        exit();
    }
    mutex_unlock(mu);                      /*释放互斥锁 */
}
```

变量mu就是互斥锁。第一个执行mutex_lock()的线程会先获得互斥锁。其他想要获得互斥锁的线程必须等待，直到第一个线程执行到mutex_unlock()释放互斥锁，才可以获得互斥锁，并继续执行线程。因此线程在进行mutex_lock()和mutex_unlock()之间的操作时，不会被其他线程影响。

每个线程必须遵守互斥锁的上述使用规则，才能保证互斥锁发挥作用。如果某个线程不尝试获得互斥锁，而是直接修改变量i，那么互斥锁就失去了保护资源的意义。互斥锁的效力在于多线程共同遵守规则，它

本身并不能硬性阻止线程对i的修改。总之，互斥锁机制需要程序员自己写出完善的程序来发挥互斥锁的功能。下面介绍的其他机制也是如此。

2. 条件变量

条件变量是另一种常用的变量。它也常常被保存为全局变量，并和互斥锁合作。举个例子，老板请了100个工人，让每个工人负责装修一个房间。当有10个房间装修完成的时候，老板就会去检查已经装修好的10个房间，然后通知这10个工人一起去喝啤酒。

我们可以并发地装修，也就是开100个线程，让每个线程对应一位工人的工作。但在多线程条件下，会有竞态条件的问题。其他工人有可能会在该工人装修好房子和检查之间完成工作。采用下面方式可以解决这个问题。

/*变量 mu 是全局的互斥锁，变量 cond 是全局的条件变量，变量 num 也是一个全局变量，用于计数*/

```
mutex_lock(mu)

num = num + 1;                                /*该工人建造房间*/

if (num <= 10) {                               /*该工人是前 10 成的*/
    cond_wait(mu, cond);                       /*等待*/
    printf("drink beer");
}
else if (num == 11) {                          /*该工人是第 11 位完成*/
    cond_broadcast(mu, cond);                 /*通知前面等待的工人 */
}

mutex_unlock(mu);
```

上面使用了条件变量。条件变量cond除了要和互斥锁mu配合之外，还需要和另一个全局变量num配合。这里的num表示装修好的房间数。这个全局变量用来构成所谓的“条件”。具体思路如下。我们在工人装修好房间，也就是执行num=num+1之后，去检查已经装修好的房间数是否小于10。由于互斥锁被锁上，所以不会有其他工人在此期间装修房间，也就是改变num的值。如果该工人是前10个完成的人，那么就调用cond_wait()函数。

`cond_wait()`做两件事情，一个是释放互斥锁`mu`，从而让别的工人可以建房；另一个是等待条件变量`cond`的通知。这样，符合条件的线程就开始等待。当“第10个房间已经修好”的通知到达时，`condwait()`会再次锁上`mu`。线程的恢复运行，执行下一句代表喝啤酒的`printf("drink beer")`。从这里开始，直到`mutex_unlock()`，就构成了另一个互斥锁结构。

前面10个调用`cond_wait()`的线程如何得到通知呢？我们注意到`else if`，即修建好第11个房间的人负责调用`cond_broadcast()`。这个函数会给所有调用`cond_wait()`的线程发通知，以便让前面10个等待的线程恢复运行。

条件变量特别适用于多个线程共同等待某个条件发生的情况。如果不使用条件变量，那么每个线程就需要不断尝试获得互斥锁并检查条件是否发生，这样大大浪费了系统的资源。

3.读写锁

读写锁与互斥锁非常相似，但它对读写做出了区分。如果一个共享资源只有读取而没有写入操作，那么多个任务可以同时读取，而不用担心竞态条件的发生。一旦有一个进程开始写入，那么其他想要读取和写入的进程必须等待该进程完成写入，才能继续操作。因此，读写锁中包含了两把锁，即读锁（R）和写锁（W）。

应用程序应该用R锁来控制读取操作。如果一个线程获得R锁，读写锁允许其他线程继续获得R锁，而不必等待该线程释放R锁。也就是说，多个进程可以同时读取同一资源。W锁用来控制写入操作，同一时间只能有一个线程获得W锁。不过，在获得W锁之前，线程必须等到所有持有共享读取锁的线程释放掉各自的R锁，以免自己的写入操作干扰到其他线程的读取。

我们这一部分介绍了一些常见的多线程同步方法。多进程同步的方法也类似，这里不再赘述。我们看到，多任务同步的实施有赖于程序员的编程付出，但在多核计算机和多主机集群流行的大背景下，多任务程序又是提高资源利用率的关键手段，因此多任务同步就变得极为重要。

第27章 进程调度

进程是一个虚拟出来的概念，用来组织计算机中的任务。但随着进程被赋予越来越多的任务，进程好像有了真实的生命，它从诞生就随着CPU时间执行，直到最终消失。不过，进程的生命都得到了操作系统内核的关照。就好像疲于照顾几个孩子的母亲，内核必须做出决定，如何在进程间分配有限的计算资源，最终让用户获得最佳的使用体验。内核中安排进程执行的模块称为**调度器**（Scheduler）。本章将介绍调度器的工作方式。

27.1 进程状态

调度器可以切换**进程状态**（Process State）。一个Linux进程从被创建到死亡，可能会经过很多种状态，比如执行、暂停、可中断睡眠、不可中断睡眠、退出等。我们可以把Linux下繁多的进程状态，归纳为三种基本状态，如图27-1所示。

- **就绪**（Ready）：进程已经获得了CPU以外的所有必要资源，如进程空间、网络连接等。就绪状态下的进程等到CPU，便可立即执行。
- **执行**（Running）：进程获得CPU，执行程序。
- **阻塞**：当进程由于等待某个事件而无法执行时，便放弃CPU，处于阻塞状态。

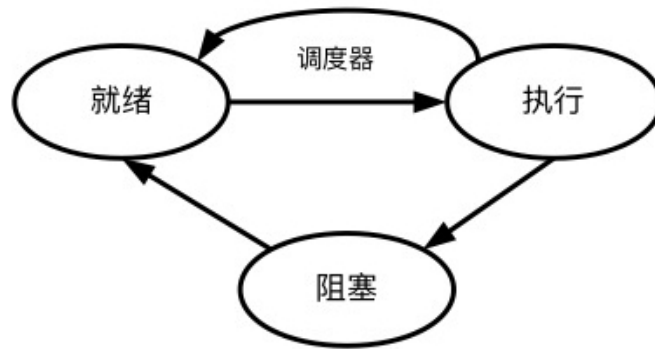


图27-1 进程的基本状态

进程创建后，就自动变成了就绪状态。如果内核把CPU时间分配给该进程，那么进程就从就绪状态变成了执行状态。在执行状态下，进程执行指令，最为活跃。正在执行的进程可以主动进入阻塞状态，比如这个进程需要将一部分硬盘中的数据读取到内存中。在这段读取时间里，进程可以主动进入阻塞状态，让出CPU。当读取结束时，计算机硬件发出信号，进程再从阻塞状态恢复为就绪状态。进程也可以被迫进入阻塞状态，比如接收到SIGSTOP信号。

调度器是CPU时间的管理员。Linux调度器需要负责做两件事：一件事是选择某些就绪的进程来执行；另一件事是打断某些执行中的进程，让它们变回就绪状态。不过，并不是所有的调度器都有第二个功能。有的调度器的状态切换是单向的，只能让就绪进程变成执行状态，不能把正在执行中的进程变回就绪状态。支持双向状态切换的调度器被称为**抢占式**（Pre-Emptive）调度器。

调度器在让一个进程变回就绪时，就会立即让另一个就绪的进程开始执行。多个进程接替使用CPU，从而最大效率地利用CPU时间。当然，如果执行中进程主动进入阻塞状态，那么调度器也会选择另一个就绪进程来消费CPU时间。所谓的**上下文切换**（Context Switch）就是指进程在CPU中切换执行的过程。内核承担了上下文切换的任务，负责储存和重建进程被切换之前的CPU状态，从而让进程感觉不到自己的执行被中断。应用程序的开发者在编写计算机程序时，就不用专门写代码处理上下文切换了。

27.2 进程的优先级

调度器分配CPU时间的基本依据，就是进程的优先级。根据程序任务性质的不同，程序可以有不同的执行优先级。根据优先级特点，我们可以把进程分为两种类别。

- **实时进程**（Real-Time Process）：优先级高、需要尽快被执行的进程。它们一定不能被普通进程所阻挡，例如视频播放、各种监测系统。

- **普通进程**（Normal Process）：优先级低、更长执行时间的进程。例如文本编译器、批处理一段文档、图形渲染。

普通进程根据行为的不同，还可以被分成**互动进程**（Interactive Process）和**批处理进程**（Batch Process）。互动进程的例子有图形界面，它们可能处在长时间的等待状态，例如等待用户的输入。一旦特定事件发生，互动进程需要尽快被激活。一般来说，图形界面的反应时间是50到100毫秒。批处理进程没有与用户交互的，往往在后台被默默地执行。

实时进程由Linux操作系统创造，普通用户只能创建普通进程。两种进程的优先级不同，实时进程的优先级永远高于普通进程。进程的优先级是一个0到139的整数。数字越小，优先级越高。其中，优先级0到99留给实时进程，100到139留给普通进程。

一个普通进程的默认优先级是120。我们可以用命令nice来修改一个进程的默认优先级。例如有一个可执行程序叫app，执行命令：

```
$nice -n -20 ./app
```

命令中的“-20”指的是从默认优先级上减去20。通过这个命令执行app程序，内核会将app进程的默认优先级设置成100，也就是普通进程的最高优先级。命令中的“-20”可以被换成-20至19中任何一个整数，包括-20 和 19。默认优先级将会变成执行时的**静态优先级**（Static Priority）。调度器最终使用的优先级根据的是进程的动态优先级，公式如下所示。

$$\text{动态优先级} = \text{静态优先级} - \text{Bonus} + 5$$

如果这个公式的计算结果小于100或大于139，将会取100到139范围内最接近计算结果的数字作为实际的动态优先级。公式中的Bonus

是一个估计值，这个数字越大，代表着它可能越需要被优先执行。如果内核发现这个进程需要经常跟用户交互，将会把Bonus值设置成大于5的数字。如果进程不经常跟用户交互，那么内核将会把进程的Bonus设置成小于5的数。

27.3 $O(n)$ 和 $O(1)$ 调度器

下面介绍Linux的调度策略。最原始的调度策略是按照优先级排列好进程，等到一个进程运行完了再运行优先级较低的一个，但这种策略完全无法发挥多任务系统的优势。因此，随着时间推移，操作系统的调度器也多次进化。

先来看Linux 2.4内核推出的 $O(n)$ 调度器。 $O(n)$ 这个名字，来源于算法复杂度的大O表示法。大O符号代表这个算法在最坏情况下的复杂度。字母n在这里代表操作系统中的活跃进程数量。 $O(n)$ 表示这个调度器的时间复杂度和活跃进程的数量成正比。

$O(n)$ 调度器把时间分成大量的微小**时间片**（Epoch）。在每个时间片开始的时候，调度器会检查所有处在就绪状态的进程。调度器计算每个进程的优先级，然后选择优先级最高的进程来执行。一旦被调度器切换到执行，进程可以不被打扰地用尽这个时间片。如果进程没有用尽时间片，那么该时间片的剩余时间会增加到下一个时间片中。

$O(n)$ 调度器在每次使用时间片前都要检查所有就绪进程的优先级。这个检查时间和进程中进程数目n成正比，这也正是该调度器复杂度为 $O(n)$ 的原因。当计算机中有大量进程在运行时，这个调度器的性能将会被大大降低。也就是说， $O(n)$ 调度器没有很好的可拓展性。 $O(n)$ 调度器是Linux 2.6之前使用的进程调度器。当Java语言逐渐流行后，由于Java虚拟机会创建大量进程，调度器的性能问题变得更加明显。

为了解决 $O(n)$ 调度器的性能问题， $O(1)$ 调度器被发明了出来，并从Linux 2.6内核开始使用。顾名思义， $O(1)$ 调度器是指调度器每次选择要执行的进程的时间都是1个单位的常数，和系统中的进程数量无关。这样，就算系统中有大量的进程，调度器的性能也不会下降。

O(1)调度器的创新之处在于，它会把进程按照优先级排好，放入特定的数据结构中。在选择下一个要执行的进程时，调度器不用遍历进程，就可以直接选择优先级最高的进程。

和O(n)调度器类似，O(1)也是把时间片分配给进程。优先级为120以下的进程时间片为：

$$(140 - \text{priority}) \times 20 \text{ 毫秒}$$

优先级120及以上的进程时间片为：

$$(140 - \text{priority}) \times 5 \text{ 毫秒}$$

O(1)调度器会用两个队列来存放进程。一个队列称为活跃队列，用于存储那些待分配时间片的进程。另一个队列称为过期队列，用于存储那些已经享用过时间片的进程。O(1)调度器把时间片从活跃队列中调出一个进程。这个进程用尽时间片，就会转移到过期队列。当活跃队列的所有进程都被执行过后，调度器就会把活跃队列和过期队列对调，用同样的方式继续执行这些进程。

上面的描述没有考虑优先级。加入优先级后，情况会变得复杂一些。操作系统会创建140个活跃队列和过期队列，对应优先级0到139的进程。一开始，所有进程都会放在活跃队列中。操作系统从优先级最高的活跃队列开始依次选择进程来执行，如果两个进程的优先级相同，则它们被选中的概率相同。执行一次后，这个进程会被从活跃队列中剔除。如果这个进程在这次时间片中没有彻底完成，它会被加入优先级相同的过期队列中。当140个活跃队列的所有进程都被执行完后，过期队列中将会有很多进程。调度器将对调优先级相同的活跃队列和过期队列继续执行下去。过期队列和活跃队列，如图27-2所示。

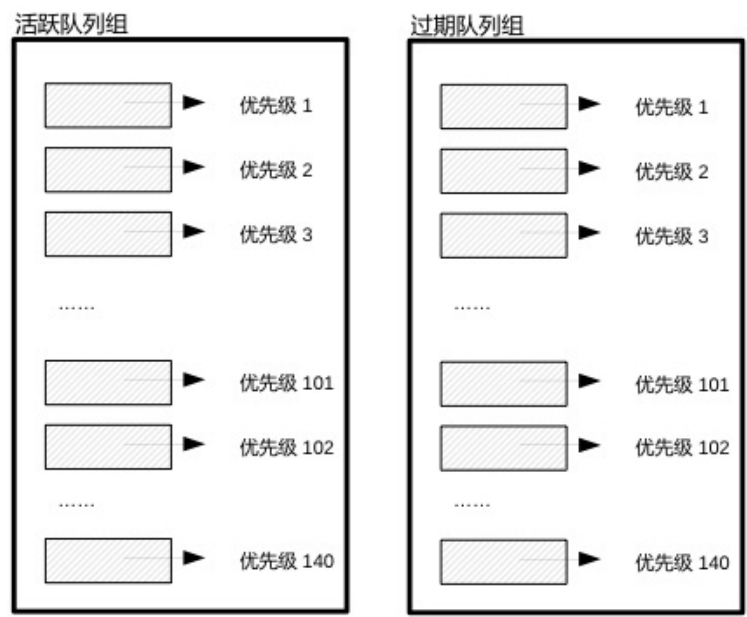


图27-2 过期队列和活跃队列（需要替换）

下面的例子有五个进程，如表27-1所示。

表27-1 进程

进程编号	优先级
A	100
B	120
C	120
D	130
E	135

Linux操作系统中的进程**队列**（Run Queue），如表27-2所示。

表27-2 进程队列

队列编号	队列内容
100	A
120	B、C
140	D
130	E

那么在一个执行周期，被选中的进程依次是先A，然后B和C，随后是D，最后是E。

注意，普通进程的执行策略并没有保证优先级为100的进程会先被执行完进入结束状态，再执行优先级为101的进程，而是在每个对调活跃和过期队列的周期中都有机会被执行，这种设计是为了避免**进程饥饿**（Starvation）。所谓的进程饥饿，就是优先级低的进程很久都没有机会被执行。

我们看到，O(1)调度器在挑选下一个要执行的进程时很简单，不需要遍历所有进程，但是它依然有一些缺点，进程的运行顺序和时间片长度极度依赖于优先级。比如，计算优先级为100、110、120、130和139这几个进程的时间片长度，如表27-3所示。

表27-3 进程的时间片长度

优先级	时间片长度（毫秒）
100	800
110	600
120	100
130	50
139	5

从表27-3中你会发现，优先级为110和120的进程的时间片长度差距比120和130之间的大了10倍。也就是说，进程时间片长度的计算存在很大的随机性。O(1)调度器会根据平均休眠时间来调整进程优先级。该调度器假设那些休眠时间长的进程是在等待用户互动。这些互动类的进程应该获得更高的优先级，以便给用户更好的体验。一旦这个假设不成立，O(1)调度器对CPU的调配就会出现问题。

27.4 完全公平调度器

从2007年发布的Linux 2.6.23版本起，**完全公平调度器**（CFS，Completely Fair Scheduler）取代了O(1)调度器。CFS调度器不对进程进行任何形式的估计和猜测。这一点和O(1)区分互动和非互动进程的做法完全不同。

CFS调度器增加了一个**虚拟运行时**（Virtual Runtime）的概念。每次一个进程在CPU中被执行了一段时间，就会增加它虚拟运行时的记

录。在每次选择要执行的进程时，不是选择优先级最高的进程，而是选择虚拟运行时最少的进程。完全公平调度器用一种叫红黑树的数据结构取代了O(1)调度器的140个队列。红黑树可以高效地找到虚拟运行最小的进程。

我们先通过例子来看CFS调度器。假如一台运行的计算机中本来拥有A、B、C、D四个进程。内核记录着每个进程的虚拟运行时，如表27-4所示。

表27-4 每个进程的虚拟运行时

进程编号	虚拟运行时（纳秒）
A	1 000
B	1 200
C	1 300
D	1 400

系统增加一个新的进程E。新创建进程的虚拟运行时不会被设置成0，而会被设置成当前所有进程最小的虚拟运行时。这能保证该进程被较快地执行。在原来的进程中，最小虚拟运行时是进程A的1000纳秒，因此E的初始虚拟运行时会被设置为1000纳秒。新的进程列表如表27-5所示。

表27-5 新的进程列表

进程编号	虚拟运行时（纳秒）
A	1 000
E	1 000
B	1 200
C	1 300
D	1 400

假如调度器需要选择下一个执行的进程，进程A会被选中执行。进程A会执行一个调度器决定的时间片。假如进程A运行了250纳秒，那它的虚拟运行时增加。而其他的进程没有运行，所以虚拟运行时不变。在A消耗完时间片后，更新后的进程列表，如表27-6所示。

表27-6 更新后的进程列表

进程编号	虚拟运行时（纳秒）
E	1 000
B	1 200
A	1 250
C	1 300
D	1 400

可以看到，进程A的排序下降到了第三位，下一个将要被执行的进程是进程E。从本质上看，虚拟运行时代表了该进程已经消耗了多少CPU时间。如果它消耗得少，那么理应优先获得计算资源。

按照上述的基本设计理念，CFS调度器能让所有进程公平地使用CPU。听起来，这让进程的优先级变得毫无意义。CFS调度器考虑到了这一点。CFS调度器会根据进程的优先级来计算一个时间片因子。同样是增加250纳秒的虚拟运行时，优先级低的进程实际获得的可能只有200纳秒，而优先级高的进程实际获得的可能有300纳秒。这样，优先级高的进程就获得了更多的计算资源。

本章中学习了调度器的基本原理，以及Linux用过的几种调度策略。调度器可以更加合理地把CPU时间分配给进程。现代计算机都是多任务系统，调度器在多任务系统中起着顶梁柱的作用。

第28章 内存的一页故事

在讨论进程时，不免要提到内存。内存是计算机的主存储器。内存为进程开辟出进程空间，让进程在其中保存数据。本章从内存的物理特性出发，深入内存管理的细节，着重介绍了虚拟内存和内存分页的概念。

28.1 内存

简单地说，内存就是一个数据货架。内存有一个最小的存储单位，大多数都是一个字节。内存用**内存地址**（Memory Address）来为每个字节的数据顺序编号。因此，内存地址说明了数据在内存中的位置。内存地址从0开始，每次增加1。这种线性增加的存储器地址称为**线性地址**（Linear Address）。我们用十六进制数来表示内存地址，比如0x00000003、0x1A010CB0。这里的“0x”用来表示十六进制。“0x”后面跟着的就是作为内存地址的十六进制数。

内存地址的编号有上限。地址空间的范围和**地址总线**（Address Bus）的位数直接相关。CPU通过地址总线来向内存说明想要存取数据的地址。以英特尔32位的80386型CPU为例，这款CPU有32个针脚可以传输地址信息。每个针脚对应了一位。如果针脚上是高电压，那么这一位是1。如果是低电压，那么这一位是0。32位的电压高低信息通过地址总线传到内存的32个针脚，内存就能把电压高低信息转换成32位的二进制数，从而知道CPU想要的是哪个位置的数据。用十六进制表示，32位地址空间就是从0x00000000到0xFFFFFFFF。

内存的存储单元采用了**随机读取存储器**（RAM，Random Access Memory）。所谓的“随机读取”，是指存储器的读取时间和数据所在位置无关。与之相对，很多存储器的读取时间和数据所在位置有关。以磁带为例，我们想听其中的一首歌，必须转动带子。如果那首歌是第一首，那么立即就可以播放。如果那首歌恰巧是最后一首，那么快进

到可以播放的位置就需要花很长时间。我们已经知道，进程需要调用内存中不同位置的数据。如果数据读取时间和位置相关，那么计算机就很难把控进程的运行时间。因此，随机读取的特性是内存成为主存储器的关键因素。

内存提供的存储空间，不仅能满足内核的运行需求，通常还能支持运行中的进程。即使进程所需空间超过内存空间，内存空间也可以通过少量拓展来弥补。换句话说，内存的存储能力和计算机运行状态的数据总量相当。内存的缺点是不能持久地保存数据。一旦断电，内存中的数据就会消失。因此，树莓派即使有了内存这样一个主存储器，也需要像SD卡这样的外部存储器来提供持久的储存空间。

28.2 虚拟内存

内存的一项主要任务就是存储进程的相关数据。我们之前已经看到过进程空间的程序段、全局数据、栈和堆，以及这些存储结构在进程运行中所起的关键作用。有趣的是，虽然进程和内存的关系如此紧密，但是进程并不能直接访问内存。在Linux下，进程不能直接读写内存中地址为0x1位置的数据。进程中能访问的地址只能是**虚拟内存地址**（Virtual Memory Address）。操作系统会把虚拟内存地址翻译成真实的内存地址。这种内存管理方式，叫作**虚拟内存**（Virtual Memory）。

每个进程都有自己的一套虚拟内存地址，用来给自己的进程空间编号。进程空间的数据同样以字节为单位，依次增加。从功能上说，虚拟内存地址和物理内存地址类似，都是为数据提供位置索引的。进程的虚拟内存地址相互独立，因此，两个进程空间可以有相同的虚拟内存地址，如0x10001000。虚拟内存地址和物理内存地址又有一定的对应关系，如图28-1所示。对进程某个虚拟内存地址的操作，会被CPU翻译成对某个具体内存地址的操作。

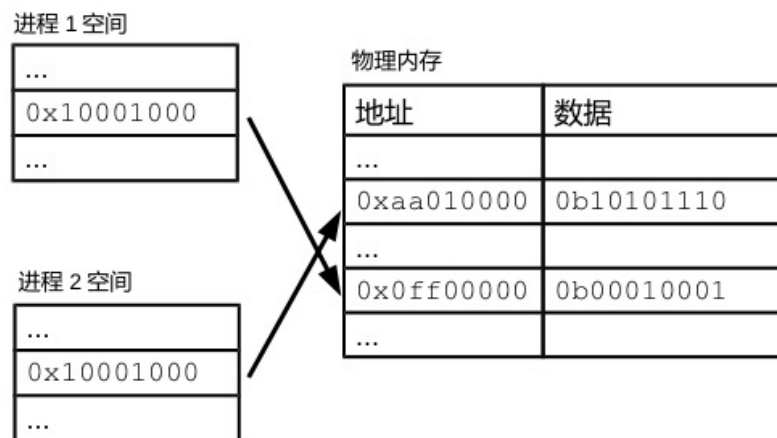


图28-1 虚拟内存地址和物理内存地址的对应

应用程序对物理内存地址一无所知。它只可能通过虚拟内存地址来进行数据读写。程序中表达的内存地址，也都是虚拟内存地址。进程对虚拟内存地址的操作会被操作系统翻译成对某个物理内存地址的操作。因为翻译的过程由操作系统全权负责，所以应用程序可以在全过程中对物理内存地址一无所知。因此，C程序中表达的内存地址，都是虚拟内存地址。比如在C语言中，可以用下面的指令来打印变量地址：

```
int v = 0;
printf("%p", (void*)&v);
```

本质上说，虚拟内存地址剥夺了应用程序自由访问物理内存地址的权利。进程对物理内存的访问，必须经过操作系统的审查。因此，掌握着内存对应关系的操作系统，也掌握了应用程序访问内存的闸门。借助虚拟内存地址，操作系统可以保障进程空间的独立性。只要操作系统把两个进程的进程空间对应到不同的内存区域，两个进程空间就成为“老死不相往来”的两个“小王国”，它们就不可能相互篡改对方的数据，进程出错的可能性也就大为减少了。

有了虚拟内存地址，内存共享也变得简单。操作系统可以把同一物理内存区域对应到多个进程空间。这样，不需要任何数据复制，多个进程就可以看到相同的数据。内核和共享库的映射，就是通过这种方式进行的。每个进程空间最初一部分的虚拟内存地址，都对应到物理内存中预留给内核的空间。这样，所有的进程就可以共享同一套内

核数据。共享库的情况也是类似的。对于任何一个共享库，计算机只需要往物理内存中加载一次，就可以通过操纵对应关系，来让多个进程共同使用。IPO中的共享内存，也有赖于虚拟内存地址。

28.3 内存分页

虚拟内存地址和物理内存地址的分离，给进程带来便利性和安全性。但虚拟内存地址和物理内存地址的翻译，又会额外耗费计算机资源。在多任务的现代计算机中，虚拟内存地址已经成为必备的设计。那么，操作系统必须要考虑清楚，如何能高效地翻译虚拟内存地址。

记录对应关系最简单的办法，就是把对应关系记录在一张表中。为了让翻译速度足够快，这个表必须加载在内存中。不过，这种记录方式的浪费惊人。如果树莓派1GB物理内存的每个字节都有一个对应记录，那么光是对应关系就远远超过内存的空间。由于对应关系的条目众多，搜索到一个对应关系所需的时间也很长，这样会让树莓派陷入瘫痪。

因此，Linux采用了**分页**（Paging）的方式来记录对应关系。所谓的分页，就是以更大尺寸的单位**页**（Page）来管理内存。在Linux中，通常每页大小为4KB。如果想要获取当前树莓派的内存页大小，可以使用命令：

```
$getconf PAGE_SIZE
```

得到结果，即内存分页的字节数：

```
4096
```

返回的4096代表每个内存页可以存放4096个字节，即4KB。Linux把物理内存和进程空间都分割成页。

内存分页可以极大地减少所要记录的内存对应关系。我们已经看到，以字节为单位的对应记录实在太多了。如果把物理内存和进程空间的地址都分成页，内核只需要记录页的对应关系，相关的工作量就

会大为减少。由于每页的大小是每个字节的4千倍，因此内存中的总页数只是总字节数的四千分之一。对应关系也缩减为原始策略的四千分之一。分页让虚拟内存地址的设计有了实现的可能。

无论是虚拟页，还是物理页，一页之内的地址都是连续的。这样一个虚拟页和一个物理页对应起来，页内的数据就可以按顺序一一对应。这意味着，虚拟内存地址和物理内存地址的末尾部分应该完全相同。大多数情况下，每一页有4096个字节。因为4096是2的12次方，所以地址最后12位的对应关系天然成立。我们把地址的这一部分称为**偏移量**（Offset）。偏移量实际上表达了该字节在页内的位置。地址的前一部分则是页编号。操作系统只需要记录页编号的对应关系。地址翻译过程如图28-2所示。

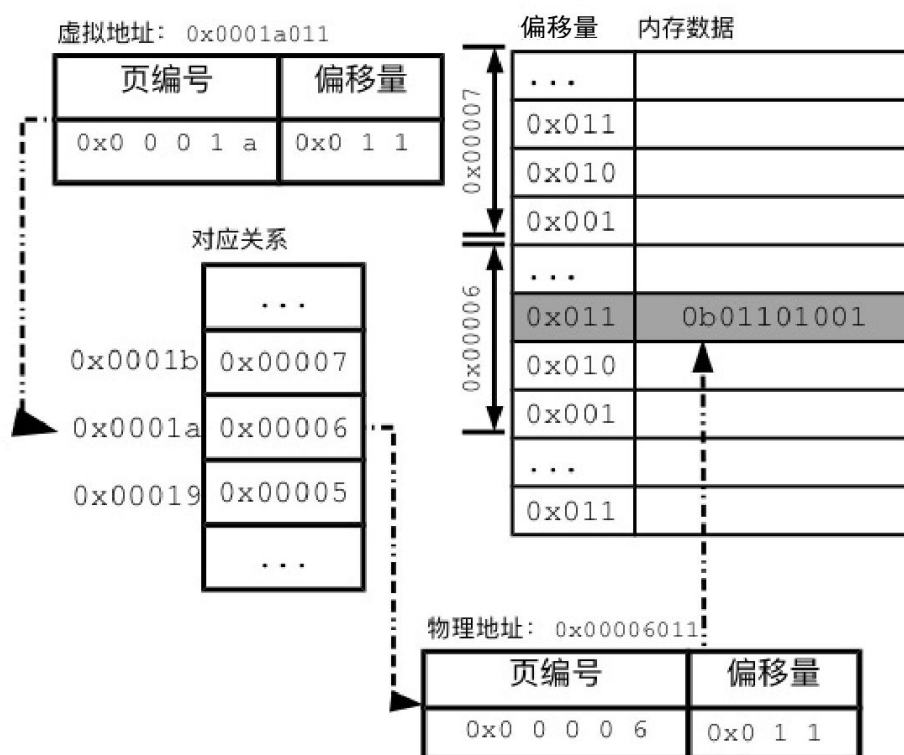


图28-2 地址翻译过程

28.4 多级分页表

内存分页制度的关键在于管理进程空间页和物理页的对应关系。操作系统把对应关系记录在**分页表**（Page Table）中。这种对应关系让上层的抽象内存和下层的物理内存分离，从而让Linux能灵活地进行内存管理。因为每个进程都有一套虚拟内存地址，所以每个进程都会有一个分页表。为了保证查询速度，分页表也会保存在内存中。分页表有很多种实现方式，最简单的一种分页表就是把所有的对应关系记录到同一个线性列表中，即如图28-2中的“对应关系”部分所示。

单一的连续分页表需要给每一个虚拟页预留一条记录的位置。对于任何一个应用进程，其进程空间真正用到的地址都相当有限。进程空间中有栈和堆。虽然进程空间为栈和堆的增长预留了地址，但是栈和堆很少会占满进程空间。这意味着，如果使用连续分页表，那么很多条目都没有真正用到。因此，Linux中的分页表采用了多层的数据结构。多层的分页表能够减少所需的空間。

我们用一个简化的分页设计来说明Linux的多层分页表。我们把地址分为了页编号和偏移量两部分，用单层的分页表记录页编号部分的对应关系。对于多层分页表来说，会进一步分割页编号为两个或更多的部分，然后用两层或更多层的多页表来记录其对应关系，如图28-3所示。

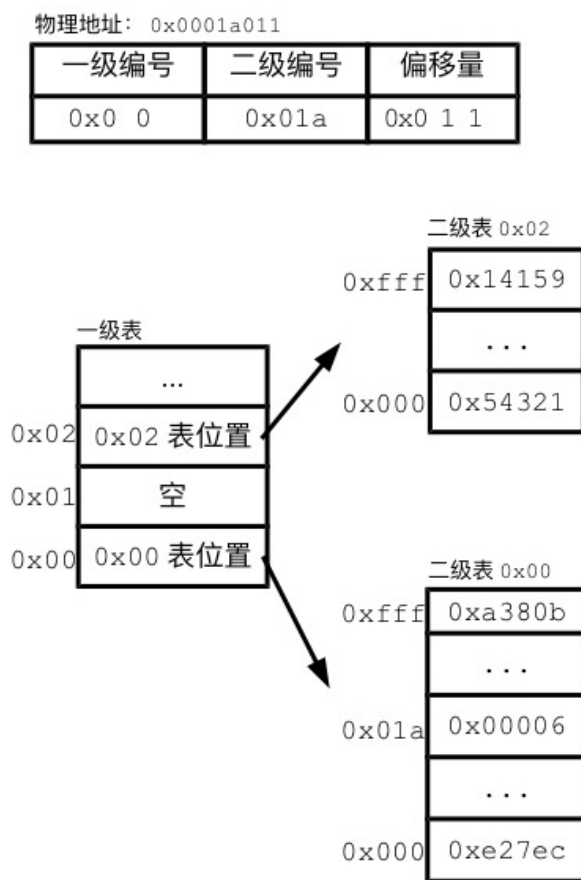


图28-3 多层分页表

在图28-3的例子中，页编号分成了两级。第一级对应了前8位页编号，用两个十六进制数字表示。第二级对应了后12位页编号，用3个十六进制编号。二级表记录有对应的物理页，即保存了真正的分页记录。二级表有很多张，每个二级表分页记录对应的虚拟地址前8位都相同。比如二级表0x00里面记录的前8位都是0x00。翻译地址的过程要跨越两级。首先取地址的前8位，在一级表中找到对应记录，该记录会告诉我们，目标二级表在内存中的位置。然后在二级表中，通过虚拟地址的后12位，找到分页记录，最终找到物理地址。

多层分页表就好像把完整的电话号码分成区号。我们把同一地区的电话号码，以及对应的人名记录在同一个本子上，再用一个上级本子记录区号和各个小本子的对应关系。如果某个区号没有使用，那么在上级本子上把该区号标记为空。同样，一级分页表中0x01记录为空，说明了以0x01开头的虚拟地址段没有使用，相应的二级表就不需

要存在了。正是通过这一手段，多层分页表占据的空间要比单层分页表少了很多。

多层分页表还有另一个优势。单层分页表必须存在于连续的内存空间，而多层分页表的二级表，可以散布于内存的不同位置，这样操作系统就可以利用零碎空间来存储分页表。还需要注意的是，这里简化了多层分页表的很多细节。最新Linux系统中的分页表多达3层，管理的内存地址也比本章介绍的长很多。不过，多层分页表的基本原理是相同的。

本章介绍了内存以页为单位的管理方式。在分页的基础上，虚拟内存和物理内存实现了分离，从而让内核深度参与并监督内存分配，应用进程的安全性和稳定性因此大大提高。

第29章 仓库大管家

在前面的章节中，我们已经用到了Linux的文件系统。通过文件系统，可以找到文件、新建文件、删除文件、读写文件。这些高层抽象的用户操作，完全可以满足日常需求。但对于Linux程序员和资深用户来说，只有知道了外部存储器的组织方式，才能深入Linux系统编程。

29.1 外部存储设备

文件系统的终极目标是把大量数据有组织地放入外部存储设备中，比如树莓派的SD卡上。以SD卡作为外部存储器的计算机并不常见。在非树莓派的PC上，更常见的外部存储器是磁盘。外部存储设备的容量一般也比内存大。它们还可以持久地保存数据，储存的数据不会随着断电而消失。外部存储器的读写速度要比内存慢。道理很简单，如果外部存储器读写速度比内存还快，那么人们会直接选用外部存储器来作为主存储器。

传统的机械式磁盘在进行随机读写时，效率会比连续读写更低。机械式磁盘由多个盘片和磁头组成，每个盘片上有多个可以存储数据的磁道。如果读写的区域不连续，磁盘需要改变磁头位置来切换磁道。在进行随机读写时，数据存活区域可能散布于不同的磁道，因此磁道切换会让读写效率大为降低。因为SD卡没有类似的机械结构，所以随机读写和连续读写的速度差距不像磁盘那么大。

再来看外部存储器中的数据组织方式。Linux通过文件系统来管理外部存储器。文件系统有很多种分类。在Linux下常见的有ext2fs、ext3fs和ext4fs。Windows系统采用的是FAT文件系统。NTFS是常用于网络存储器的文件系统。每种文件系统都有自己的一套数据管理策略，自然也会各有优缺点。但无论是哪种文件系统，都至少应该有三方面的功能。

- (1) 通过名字和层级来组织文件，比如文件名和路径。

(2) 提供操作文件的接口，比如查找、新建、删除、读取和写入文件。

(3) 提供权限功能，比如文件保护和文件共享。

同一个外部存储器可以划分成一个或多个**分区**（Partition），每个分区可以用一种文件系统格式来管理。以树莓派为例，在SD卡上烧录Raspbian镜像后，SD卡的存储空间就会划分成两个分区。一个空间是启动分区，采用了FAT32形式的文件系统，启动分区主要用于开机启动，空间较小。剩下的空间是主分区，采用了ext4fs形式的文件系统。

29.2 外部存储器的挂载

SD卡上的两个分区采用了两种文件系统。但最终在运行的Raspbian上，只会看到一个以根目录/为起点的文件系统。也就是说，两个物理分区以某种形式合并到了Linux的文件树上。我们把这个过程称为**挂载**（Mounting），即让文件树上的某个目录和存储器的物理分区对应起来。这个目录称为**挂载点**（Mounting Point）。从挂载点开始向下的子文件树，实际上就对应了挂载的物理分区。

Linux系统的挂载信息都保存在文件 */etc/fstab* 中。查看树莓派中该文件的记录：

```
proc          /proc          proc defaults      0      0
/dev/mmcblk0p6 /boot          vfat  defaults        0      2
/dev/mmcblk0p7 /              ext4   defaults,noatime  0      1
```

可以看到，树莓派SD卡的主分区挂载在根目录/上，而启动分区挂载在根目录下的 */boot* 上。这两棵文件树有重叠的从属关系，以根目录/为起点的文件树，包括了以 */boot* 为起点的文件树。这种情况下，Linux以子文件树优先，把启动分区挂载在 */boot* 上。主分区的挂载点是根目录/，但不再包括 */boot* 子文件树。因此，*/boot* 下的数据都会存放于启动分区，其他的数据存放于主分区。

一个外部存储器必须经过挂载，才能加入操作系统的文件树。也只有加入文件树后，应用程序才能通过文件系统来访问外部存储器中的数据。在树莓派中插入一个U盘，用fdisk命令可以找到这个U盘。

```
$sudo fdisk -l
```

它会自动挂载到 */media* 下的一个目录上，例如 */media/MyUsbDrive*。因此，我们往 */media/MyUsbDrive* 存入的文件，实际上都会存入U盘。如果对系统自动分配的挂载点不满意，那么可以卸载U盘，再挂载到一个自定义的挂载点。首先，卸载U盘，假如USB设备位于 */dev/sda1*，那么卸载U盘的命令就是：

```
$sudo umount /dev/sda1
```

然后，设置设备的默认挂载点，默认挂载点的配置文件在 */etc/fstab* 上。使用nano工具来编辑这个文件：

```
$sudo nano /etc/fstab
```

这个文件里的每一行都是一个设备的挂载设置，例如下面这一行：

```
/dev/mmcblk0p7 /      ext4    defaults,noatime 0      1
```

这里有6个参数，含义如下。

- (1) 设备名称。一般是 */dev/xxx*。
- (2) 挂载点。对于USB设备默认是 */media/xxx*。
- (3) 文件系统格式。例如ext4。
- (4) 设备参数。例如defaults、noatime。
- (5) 一个不再使用的参数，通常设置为0。

(6) 磁盘检测设置。1为根文件系统，2为永久挂载磁盘，0为可插拔的移动设备。

挂载完成后，可以用df命令来查询文件系统的挂载情况：

```
$sudo df
```

存储器开始部分的块会有一个总的**分区表**（Partition Table），记录着存储器的基本信息，比如**块**（Block）的大小、存储器的编号和可用空间。块是存储器的读写单元。即使一个文件小于一个块，它还是会占据这个块的完整空间。此外，分区表中还逐项记录每个分区的信息，包括分区的起始位置和大小。随后的存储空间划分成分区。不同的分区可以采用不同的文件系统。

29.3 ext文件系统

根据文件系统类型的不同，分区有不同的存储格式。先来看树莓派ext4格式的主分区。ext4全称是**第四代扩展文件系统**（Fourth Extended File System）。从ext到ext4的文件系统，都是专门为Linux内核开发的。相对于第一代的ext来说，ext4增加了许多高级特征。但这四代操作系统的共同特色是围绕inode来组织文件。笔者也将围绕inode来展示ext系列的文件系统，并有意忽略一些高级特征。

对于一个ext分区来说，内容可以分为如图29-1所示的几个部分。装有操作系统的ext分区的第一个块是**引导块**（Boot Block）。引导块中有**引导加载程序**（Boot Loader），帮助计算机在开机时加载Linux内核。引导加载程序储存有内核的相关信息，比如内核名称和内核所在位置。但在树莓派中，FAT32的启动分区负责开机启动。树莓派的引导加载程序也在启动分区。因此，树莓派上的ext主分区并没有引导块。

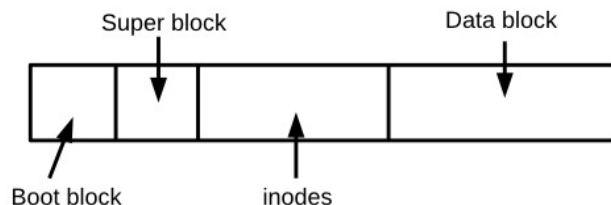


图29-1 ext分区

每个ext分区会有一个**超级块**（Super Block）。超级块中记录着文件组织的信息，包括文件系统的类型、inode的数目、块的总数和空闲

数量等。超级块对于文件系统来说至关重要。如果超级块损坏，则会导致整个分区的文件系统损坏。

超级块后面跟着inode表和**数据块**（Data Block）部分。一个inode表中有多个inode。所谓的inode，是描述文件存储信息的数据结构。文件有一个对应的inode。每个inode有一个唯一的**整数编号**（Inode Number）。在Linux下，可以使用命令来查询文件的inode编号。

```
$stat example.txt
```

一个文件除了自身的数据之外，还会有附属信息。附属信息包括文件大小、拥有人、拥有组、修改日期等。这些附属信息就存在inode中。因此，inode对于文件管理和文件安全都很重要。

除了这些附属信息，inode还存有文件包含的所有数据块的位置信息。这些位置信息被称为指向数据块的指针。在Linux系统中，一个大文件可以分成几个数据块存储，就好像是分散在各地的龙珠。为了顺利地集齐龙珠，我们需要地图的指引。当Linux想要打开一个文件时，必须先找到文件对应的inode，然后根据inode这张地图的指引将所有数据块收集起来，才能拼凑出一个完整的文件。在Linux中，我们通过解析路径，根据沿途的目录文件来找到某个文件。目录文件的每个条目对应了一个子文件的文件名，以及该文件的inode编号。

以 */var/test.txt* 文件的存储为例，假设其存储结构如图29-2所示。当我们输入代码`$cat/var/test.txt`时，Linux将在根目录文件中找到 */var* 这个目录文件的inode编号10747905，然后根据inode中指针指向的数据块合成出 */var* 目录文件。随后，Linux重复上述过程，根据 */var* 中 *test.txt* 文件的inode编号10749034，找到 *test.txt* 的数据。

当写入一个文件时，Linux会分配一个空白inode给该文件，将其inode编号记入该文件所属的目录，然后选取空白的数据块，让inode指针指向这些数据块，并向内存中放入数据。

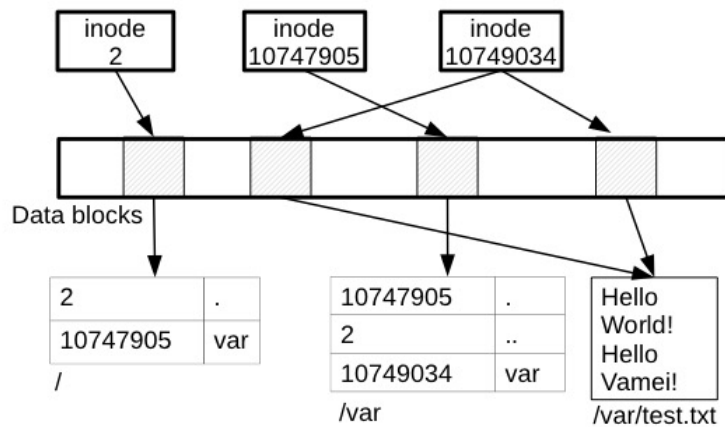


图29-2 `/var/test.txt` 相关文件存储

29.4 FAT文件系统

再来看FAT32格式的树莓派启动区。正如上面提到的，这个启动区有一个引导块，用于在开机时加载Linux内核。引导块之后是**文件分配表**（FAT，File Allocation Table）。文件分配表的组织形式和inode不同，但起到了和inode类似的作用。

FAT32是FAT文件系统家族的一员。FAT文件系统其实就是以“文件分配表”的英文简写来命名的，由此可见文件分配表对于FAT文件系统的重要性。文件分配表按照顺序对应了所有的数据块。在文件分配表的一条记录中，说明了同一个文件中下一个数据块的位置。比如，文件分配表的第2条记录中记录了5，这就说明了2号数据块的下一个数据块是5号数据块。当一个数据块是文件的最后一个数据块时，它就不再有下一个数据块了。它在文件分配表中的记录，也会填写成固定的0xffff。只要知道了一个文件的起点数据块位置，就能根据文件分配表找到该文件的所有数据块。

此外，FAT文件系统还有一个区域专门记录FAT根目录信息。其他的子目录则以文件的形式保持。目录中的每条记录对应了一个文件，除了文件名和文件属性，还记录了文件的起始数据块的位置。从根目录出发，我们可以通过记录中起始数据块的位置，配合文件分配表来组装根目录下的子目录和文件。依此类推，我们可以找到整个FAT文件的文件。

由此可见，ext的组织形式着眼于文件，因此以inode为主要的中间层。而FAT的组织形式着眼于数据块，所以以文件分配表为主要的中间层。FAT的文件分配表的记录总数和数据块总数相同，可能会很占空间。此外，FAT必须按照顺序一个一个找数据块，没法像ext那样从inode中获得一张完整的地图。因此，当文件在存储器上比较零散时，FAT没法像ext一样优化读写路径，但由于Windows系统的成功，FAT文件系统的应用依然非常广泛。

29.5 文件描述符

我们已经从底层了解了文件的存储方式，现在自上而下地看程序打开文件的过程。在Linux的应用程序中，当我们打开一个文件时，会获得一个整数来代表该文件。这个整数称为**文件描述符**（File Descriptor）。

进程描述符中有一个文件描述符表，记录了该进程所有已经打开的文件。文件描述符说明了目标文件在文件描述符表中的位置。文件描述符表的每条记录中包含一个指针。有趣的是，这个指针并没有直接指向文件的inode，而是指向了一个**文件表**（File Table）。文件表中的指针指向加载到内存中的inode，也就是目标文件的inode。也就是说，从文件描述符出发，首先找到文件描述符表中的记录，再找到文件表，然后找到文件的inode，最后抵达数据块。一个进程打开了两个文件，如图29-3所示。

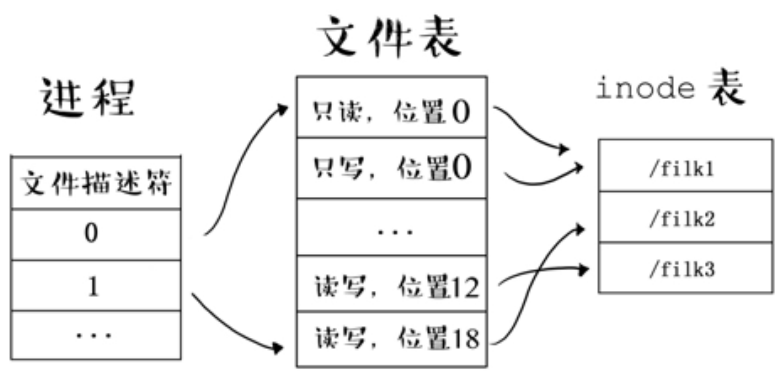


图29-3 进程、文件表与inode表

每个文件表中记录着**状态标识**（Status Flag），比如只读、读写等。文件打开状态是在打开文件时由应用程序决定的。文件表中还记录了当前读写位置。当有两个进程打开同一个文件时，每个进程都会有一个文件表。因此，即使是两个进程同时打开一个文件，在不同的进程中，同一文件会有不同的状态和读写位置。

注意进程fork对文件描述符的影响。当进程fork时，子进程将只复制文件描述符表。子进程文件描述符表中的指针，还是指向父进程的文件表。这样父进程和子进程将共享文件打开状态和读写位置。当父进程和子进程同时操作已打开文件时，有可能会相互干扰，在这种情况下编写程序要特别小心。

第30章 鸟瞰文件树

第29章自下而上地介绍了外部存储器的底层细节。本章将自上而下，鸟瞰完整的Linux文件树。直接从属于根目录 `/` 的文件和目录都是系统必备的关键内容。我们来看它们的功能。

30.1 `/boot`和树莓派启动

`/boot` 下挂载了FAT32格式的启动分区，里面的文件用于树莓派的开机启动。计算机启动是一个神秘而有趣的过程，先来看计算机常见的启动方式。

当我们打开一台普通计算机的电源时，计算机一般会从主板的BIOS上读取其中所存储的程序。BIOS知道直接连接在主板上的硬件。它从默认存储设备中读取最开始512字节的数据，即所谓的MBR（Master Boot Record）。用户也可以通过BIOS配置，从其他数据存储设备中找到MBR。通过MBR，计算机知道要从该存储设备的哪个分区来找引导加载程序。引导加载程序储存有操作系统的相关信息，比如操作系统名称、内核所在位置等。随后引导加载程序加载内核，操作系统开始工作。

树莓派的开机方式有别于普通计算机。树莓派是一块集成电路板，没有主板，也没有BIOS。树莓派电路板上携带着启动程序。板载启动程序会挂载FAT32的启动分区，并运行其中的引导程序 `bootcode.bin`。它负责下一阶段的启动工作，会从SD卡上找到GPU固件 `start.elf`，将固件写入GPU。GPU在 `start.elf` 的指挥下，会读取系统配置文件 `config.txt` 和内核配置文件 `cmdline.txt`，并启动内核文件 `kernel.img`。当内核加载成功时，处理器开始工作，系统启动正式开始。

普通计算机的启动流程有更多可选项。而树莓派通过板载程序来固化启动流程，让启动更可控。但无论是哪种开机流程，我们看到操作系统是通过小程序加载大程序，并相继唤醒硬件的过程。内核加载成功之后，操作系统正式开始工作。Linux系统还会进行一系列的准备工作，来让操作系统更好用。

内核会首先预留运行所需的内存空间，然后通过驱动程序检测计算机硬件。这样，操作系统就可以知道自己有哪些硬件可用。随后，内核会启动init进程。到此，内核完成了启动阶段的工作。进程init会运行一系列初始脚本，这些脚本用于准备操作系统。

- 设置计算机名称、时区等。
- 检测存储器。
- 挂载存储器。
- 清空临时文件。
- 设置网络。
- 启动其他后台进程。

这些初始脚本运行完毕，操作系统就准备好了，只是，还没有人可以登录。进程init会给出登录对话框，或是图形化的登录界面。登录之后，就是Shell或图形化的用户界面了。

30.2 应用程序相关

在Linux系统中，应用程序都编译成二进制的可执行文件，位于名为 **bin** 的目录下。“bin”就是二进制“binary”的简写，根目录下就有 **/bin**。这里保存着Linux系统运行必须的应用程序。

bash	cat	chmod	cp	date	echo
expr	kill	ln	ls	mkdir	mv
pwd	rm	rmdir	sleep	test	unlink

fdisk	hwclock	ifconfig	reboot	shutdown
-------	---------	----------	--------	----------

根目录在 **/sbin** 下保存了系统启动、修复和恢复所必需的应用程序。**/usr** 下有一个 **/usr/bin** 目录，也存放了可执行文件。**/usr/bin** 保存了次要一些的应用程序。在大多数Linux发行版本中，**/usr/bin** 中包含的应用程序比 **/bin** 中多得多。Raspbian中apt-get

安装的程序，大多也会出现在这里。虽然 **/usr/bin** 程序对于Linux程序的运行不是那么关键，但很可能是常用的应用程序，比如文本编辑程序、编译器、数据库等。我们已经接触过很多

/usr/bin 中的程序，比如：

cut	diff	free	gcc	head	locate
man	make	nano	nice	sftp	ssh
tail	uniq	vi	wc	which	who

同理， **/usr/sbin** 保存的也是次要的系统维护程序，比如任务规划程序cron。最后， **/usr/local/bin** 和 **/usr/local/sbin** 也是保存应用程序的地方。这里通常保存了用户自己编写或手动编译安装的应用程序。

虽然计算机可以直接理解二进制可执行文件，但是二进制可执行文件往往还要依赖其他的文件才能正常运行。这些文件包含了分布于 **/lib**、**/usr/lib**、**/usr/local/lib** 中的库。这些编译好的库让程序可以复用已经成熟的代码，从而实现更加强大的功能。此外， **/usr/include** 和 **/usr/local/include** 中有头文件。当程序跨文件执行库里的函数时，也需要引用包含了该函数声明的头文件。

30.3 /etc与配置

/etc 中有很多配置文件。这些配置文件可以影响系统和应用程序的行为。我们在之前已经见过很多 **/etc** 下的配置文件，借助这些已经见过的文件来说明 **/etc** 下文件的类型。

/etc 保存着关键的操作系统配置文件，这些配置文件可以改变操作系统级别的行为，如表30-1所示。

表30-1 路径与功能

路 径	功 能
/etc/default/locale	本地设置，比如语言、字符编码
/etc/default/keyboard	键盘设置
/etc/localtime	时间与时区配置
/etc/modules	可加载模块配置

操作系统启动时的init进程及init调用的脚本也在 */etc* 下。这些脚本在启动阶段执行，并最终决定呈现给我们的操作系统。路径与功能如表30-2所示。

表30-2 路径与功能

路 径	功 能
<i>/etc/init.d/</i>	初始化相关文件
<i>/etc/rc.local</i>	初始化脚本

我们在Linux用户中看到， */etc* 保存着用户和用户组的相关信息。增加或删除用户的操作，实际上就是修改这些文件，如表30-3所示。

表30-3 路径与功能

路 径	功 能
<i>/etc/passwd</i>	用户列表
<i>/etc/group</i>	用户组列表
<i>/etc/passwd</i>	用户密码

上面提到的配置文件都是操作系统级别的。 */etc* 不仅有操作系统级别的配置文件，还包括了应用程序的配置文件。这些配置文件是全局的，对所有用户都有效，如表30-4所示。

表30-4 路径与功能

路 径	功 能
<i>/etc/motion/motion.conf</i>	Motion 的配置文件
<i>/etc/apt/sources.list</i>	apt-get 软件源配置
<i>/etc/wpa_supplicant/wpa_supplicant.conf</i>	Wi-Fi 设置

应用程序也可以有自己的初始化脚本，如表30-5所示。

表30-5 路径与功能

路 径	功 能
<i>/etc/bashrc</i>	Motion 初始化设置
<i>/etc/nanorc</i>	Nano 初始化设置
<i>/etc/virc</i>	Vi 初始化设置

30.4 系统信息与设备

内核直接管理的硬件信息可以在 */proc* 下查询。*/proc* 其实是一个虚拟文件系统，直接对应了内存上的内核空间。通过 */proc*，内核给用户提供了一个查询内核信息的简易窗口^[1]。*/proc/cpuinfo* 中保存着CPU信息，*/proc/meminfo* 中保存着内存使用信息。因为内核直接管理的设备对于计算机运行至关重要，所以 */proc* 下的文件大多是只读的，不允许用户直接进行写入操作。内核还保存着进程的信息。这些原本在内核空间的信息也以文件的形式呈现在 */proc* 目录下。

/dev 目录中保存着设备文件。每个设备文件对应着一个设备，比如存储器和UART接口。通过这些设备文件，设备还可以是没有硬件实物的虚拟设备，比如终端。我们可以以文件操作的形式直接和设备进行交流，通过读写 */dev/ttyAMA0* 来与UART接口通信。

Linux 的设备有**主编号**（Major Number）和**副编号**（Minor Number）。主编号说明了设备的类型，在 */dev* 中对应为一个名字，比如“ttyAMA”。副编号就是后面跟的“0”，即该类型下编号为0的设备。通过 man 命令来找出某种设备的主编号，比如：

```
$man 4 ttyAMA
```

Linux下的*/mnt* 用于挂载额外的文件系统，比如网络硬盘、光驱和额外的硬盘。对于 */mnt* 下的存储设备，通常要手动挂载或者在挂载文件里增加对应条目。*/media* 用于挂载可插拔设备，如U盘和数码相机。这些设备插入电脑USB接口，Linux就会自动挂载在 */media* 下。近年来，随着可插拔设备的快速发展，*/media* 的使用频率超过了 */mnt*。

30.5 其他目录

本节介绍 */home*、*/var* 和 */tmp* 目录。这三个目录下的内容都和用户或应用程序的使用情况有关，目录占据的空间可能随着时间快速变化。

Linux是多用户系统，每个用户会有一个用户目录，位于不同的路径下。*/root* 是root的用户目录。该目录文件的拥有者和拥有组都是root。其

他用户的用户目录都位于 */home* 下。用户pi的用户目录位于 */home/pi*，这个目录文件的拥有者和拥有组都是pi。因为用户数据可能快速增长，所以 */home* 往往挂载有额外的存储器，拥有独立的存储空间。

/var 用于保存系统中会动态增长的数据，比如 */var/log* 下的系统日志和应用程序日志。此外，每个应用程序也会产生动态增长的数据。就拿邮件程序来说，其可执行文件是一个大小不变的静态文件，但电子邮件的相关文字和图片会随着用户使用快速增长。因此，电子邮件常常归档保存在 */var* 下。缓存数据占据的空间经常浮动变化，因此也保存在 */var* 下。由于 */var* 的动态变化性，它经常挂载有独立的存储器。

应用程序运行的过程中可能会有一些临时数据需要保存到文件系统中，比如数学运算的中间结果。如果应用程序不想持久保存这些文件，就会把这些文件放在 */tmp* 文件下。因为应用程序可能依赖这些临时文件，所以随意修改 */tmp* 下的文件可能造成应用程序的崩溃。幸好，*/tmp* 下的文件会自动清空，因此 */tmp* 下的文件基本不需要维护。不同版本的Linux系统会选择不同的时间来清空临时文件。Raspbian会在开机后清空 */tmp* 文件夹。由于临时文件的生长很难预知，因此 */tmp* 也经常位于额外的存储器中，以免临时文件和系统文件竞争空间。

我们上面看到了Linux的文件树的结构，这个文件树继承自UNIX。在几十年的发展中，虽然有缓慢演进，但是结构上并没有太大的变化。了解这个历久弥新的文件树，对于我们使用整个UNIX家族的操作系统大有裨益。

[1] 与之相对，同样保存有系统信息的 */sys* 目录就保存在磁盘上。*/sys* 只在特定用途出场，用户用到的概率比较低，比如在GPIO编程中就用到了 */sys* 目录。

第31章 分级存储

树莓派上的三种电子元件都有存储数据的功能：CPU缓存、内存和SD卡储存，如表31-1所示。三种元件的速度和容量各不相同。存储元件的容量和速度是个矛盾。为了兼顾性能和成本，计算机大多采取分级存储的形式，从而让不同速度的存储元件协同工作。分级存储的设计，兼顾了读取速度、存储容量和计算机的稳定性。

表31-1 树莓派3B型的各项存储器指标

类 别	大 小	访问速度排序	持 久
CPU	一级缓存：32KB	1（最快）	否
	二级缓存：512KB	2	
内存	1GB	3	否
SD 卡	大于 4GB	4（最慢）	是

31.1 CPU缓存

计算机把最快的存储元件用在最繁忙的地方。CPU是树莓派执行程序的核心，我们编写的程序和需要处理的各种数据都要加载到CPU中才能执行。除了CPU频率外，CPU对数据的访问速度是决定其运行速度的一大重要因素。因此，CPU上配置了两级的高速缓存，来让CPU更快地提取到数据。

由于造价昂贵，CPU缓存的容量不大。当CPU需要读写某个内存地址时，它会先检查该内存地址的数据是否已经存在于某条**缓存记录**（Cache Entry）中。如果缓存记录中的内存地址信息和CPU寻址信息相符，那就说明数据已经缓存了，这种情况叫作**缓存命中**（Cache Hit）。CPU会直接读写缓存中的目标记录，速度会比读写内存快很多。如果CPU想要读写的数据不在缓存中，就是**缓存缺失**（Cache Miss），那么缓存会增加一条新的缓存记录，把内存地址的数据加载到该缓存记录中。CPU随后从缓存中读写数据。

出于成本的原因。我们不可能把计算机的全部数据放在CPU缓存中。缓存中无法容纳的数据，只能存放于内存和SD卡这样速度较慢的存储空间中。既然这样，CPU缓存必须有一个策略，决定把哪些数据放在缓存中。为了应对缓存缺失的情况，缓存必须增加新的缓存记录。如果缓存已经没有空余的空间，则必须选择替换缓存中的一个记录。这条已经存在的缓存记录被称为**牺牲者**（Victim），新的缓存记录会被放在牺牲者所在的位置。

选择牺牲者的常见的方法有4种。

- **最少使用**（LFU，Least Frequently Used）的数据。
- **最久没有使用**（LRU，Least Recently Used）的数据。
- **最早被缓存**（FIFO，First-In First-Out）的数据。
- **随机替换**（Random Replacement）。

以LRU策略为例，学习缓存的替换策略。如果CPU采用了LRU策略，那么CPU为每个缓存记录增加一个计数。当CPU读缓存时，LRU会把命中记录的计数清零，而其他记录的计数增加1。如果一条记录长期没有被读取，那么它的计数就会越来越大。在选择牺牲者时，CPU缓存会选择计数最大的记录作为牺牲者。

上面对缓存工作流程的描述只是基于一层缓存的。实际上，树莓派中存在两级缓存。一级缓存L1的读写速度高于二级缓存L2，而二级缓存L2的速度又高于内存的速度。沿用已经讨论过的工作流程，在一级缓存和二级缓存之间、二级缓存和内存之间进行数据交互。两级缓存夹在CPU到内存之间，弥补了两者的速度差异，让计算机的数据读写变得更有效率。

类似的缓存技术在计算机中使用很广。除了数据，CPU还会缓存来自内存的指令及分页记录。很多技术的实现都离不开缓存带来的效率，以虚拟内存为例，虚拟内存技术可以实现很多功能，比如构建进程空间和实现内存共享，但虚拟内存不是免费的。内核必须记录虚拟内存页和物理内存页的对应关系，并花费额外的CPU时间进行地址转换。利用缓存技术把分页记录放在CPU内部的高速元件上，可以有效解决寻址的效率问题。

31.2 页交换

借用缓存技术，我们弥补了CPU和内存之间的速度差。而虚拟内存技术，则可以把外部存储器空间当作内存用。其实虚拟内存诞生之初，正是为了实现这一目的。

长期以来，计算机运行中一直面临一个恼人的问题：进程所需的空間有可能超过计算机的物理内存空间。这个时候计算机就无法运行对应程序了。例如，一个内存为1GB的树莓派，它的进程数据需要占据2GB的空间，那么树莓派就无法执行这个操作。虽然内存空间在不断增加，但程序所需空间也越来越庞大，内存空间成了程序发展的屏障。而在计算机上，能提供足够大存储空间，只能是外部存储器。由于内存和外部存储器的速度差异非常大，因此我们需要一种技术，在把外部存储器空间变成内存空间的同时，还能一定程度上保证计算机运行的效率。

虚拟内存可以把一部分的外部存储器空间转换成内存空间，让应用程序可以虚拟地增加内存大小。这一技术的关键在于**页交换**（Page Swap）。所谓的页交换，就是进程空间和外部存储空间以页为单位交换数据。虚拟内存是一套管理数据和数据地址的方法，也可以用于外部存储空间的管理。操作系统可以把一部分外部存储空间划分成页，称为**交换空间**（Swap Space）。操作系统按照管理内存的方式来管理交换空间。物理内存和交换空间加在一起大大拓展了实际存储容量。

当然，外部存储器读写速度慢的瓶颈始终存在。为了保证读写效率，应用程序只用在物理内存中的虚拟内存。当程序访问的数据恰好位于交换空间时，内核就会启动页交换，把交换空间的页转移到物理内存中，随后内核把分页对应到该物理内存位置，并通知应用程序继续进行数据操作。这样，程序访问的虚拟内存地址就指向了物理内存中的数据位置。对于应用程序来说，它只是根据虚拟内存地址进行操作，整个过程都不需要知道内核的幕后动作。

具体来说，内核记录着虚拟内存的对应关系。当应用进程访问虚拟内存页时，内核会根据对应关系，知道物理页存在内存还是外部存储器中。如果该页存在外部存储器，内核则会让程序短暂休息，然后将外部存储器中这一页的内容放入物理内存中。如果内存空间已满，那么虚拟内存要选择把内存中的一页移出交换空间，从而为要进入内存的页准备好空间。在这个过程中，内存和外部存储器交换了一页，这也是页交换

得名的原因。移出内存的页充当了牺牲者。其实页交换和CPU的缓存替换非常相似，选择牺牲者的方法也和缓存的替换策略一样。Linux操作系统使用了一种类似于LRU的策略，即选择最久没有使用的分页作为牺牲者。

31.3 交换空间

本节介绍交换空间的具体实施方法。交换空间有交换分区和交换文件两种形式。交换分区就是用一个独立的存储器分区作为交换空间，和一般的磁盘分区不同，它没有文件系统，完全以页的方式进行管理。交换文件是文件系统中的特殊文件，它占据的空间以页的方式进行管理，作为交换空间。我们可以使用下面的命令查看交换空间：

```
$sudo swapon -s
```

输出结果如下：

Filename	Type	Size	Used	Priority
/dev/sda5	partition	859436	0	-1

每一行列出的都是系统正在使用的交换空间。Type字段表明该交换空间是一个分区而不是文件，通过Filename可以知道交换分区是磁盘sda5。Size字段表明磁盘大小，单位是KB，Used字段是表示有多少交换空间被使用。Priority字段表示Linux系统的交换空间使用优先级。如果在Linux系统中挂载两个或更多具有相同优先级的交换空间，那么Linux会交替使用。如果两个交换空间正好位于两种设备上，那么这种交替使用的方式可以提升交换性能。用mkswap命令把分区/dev/hdb1 变成交换分区：

```
$sudo mkswap /dev/hdb1
```

用swapon命令激活交换分区：

```
$sudo swapon /dev/hdb1
```

再次确认/dev/hdb1已经加入交换空间：

```
$sudo swapon -s
```

Linux也支持交换文件形式的交换空间。在Linux环境中，创建文件比创建分区简单得多，可以用dd命令创建一个1GB的文件，比如：

```
$sudo dd if=/dev/zero of=/var/swapfile bs=1024 count=1048576
```

交换文件就是 */var/swapfile* 。选项 count 说明了文件大小，即 1048576KB。创建文件后，就可以用mkswap调用交换文件。

```
$mkswap /var/swapfile
```

激活交换文件，让交换文件成为可以使用的交换空间。

```
$swapon /var/swapfile
```

31.4 外存的缓存与缓冲

通过页交换技术，我们用外部存储器弥补了内存容量的不足，但页交换毕竟只发生在外存的交换空间。对于文件系统管理形式的其他分区，我们一样要解决内存和外存互动的问题。内存和外存速度差异巨大。因此，操作系统读写文件时，必须要想办法弥补两者之间的速度差异。

CPU缓存技术可以弥补这种差异。如果内存中的部分空间可以用来缓存常用的文件系统数据，就可以大大减少外存的访问量。这种技术就是**页缓存**（Page Cache），页缓存和CPU缓存非常类似。在读取外存中的文件时，文件的数据会先存在内存中未使用的页上。此后，如果需要读取相同的数据，那么CPU可以直接从内存提取。当内存中可用于页缓存的空间填满时，计算机使用类似于LRU的策略选择牺牲者，用新的文件数据来替换掉缓存页。通过free命令，可以查看内存中页缓存空间的大小。

```
$free
```

	total	used	free	shared	buffers	cached
Mem:	945512	765084	180428	28328	164872	328508
-/+ buffers/cache:		271704	673808			
Swap:	2097148	0	2097148			

在返回结果中，cached那一列说明了页缓存空间的大小。除了页缓存，内存还会在其他场景下使用缓存思想。比如，内存中会缓存文件系统的inode。读取文件的inode，是获得文件数据的第一步。对于频繁读写的文件，如果能在内存中缓存inode，那么文件的读写效率也会大大提高。

我们再来看另一种思想，即缓冲读写的思想。举一个例子，进程往一个文件中写入15个字符“Vamei loves RPI”，进程可以每次从内存中拿一个字符写入外存。由于外存写入速度慢，那么在外存完成这个字符写入的过程中，进程都是闲置的。当然，进程可以进入阻塞状态，把CPU出让给其他的进程。然而，进程状态切换需要付出代价。此外，外存写入字符前需要进行一些准备动作，比如找到写入块位置。分开写入15个字符，外存就要重复15次准备动作。

缓冲的目的就是在内存中收集多个待写入的字符，再一次性写入外存。这一方面减少了进程切换状态的次数。另一方面，外存可以共用同一套准备动作，从而减少开销。内存为进程打开的文件保留**缓冲区**（Buffer），用于收集待写入文件的文本。缓冲区采用先进先出的策略，先写入的字符会被先取出。在**刷新**（Flush）缓冲区时，缓冲区中存储的文本会按照先后次序一次性写入外存。操作系统的内核提供了缓冲区。因此，很多时候用write()系统写一个字符到文件，字符并没有真正存入文件。缓冲区的写入与刷新，如图31-1所示。

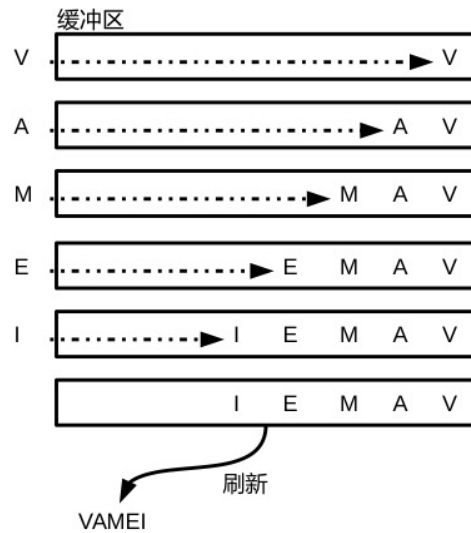


图31-1 缓冲区的写入与刷新

计算机会在多个条件下刷新缓冲区。

- 缓冲区填满了数据。
- 文件关闭。
- 进程终结。
- 文本中出现换行符。
- 该文件出现数据读取。

Linux内核中内置了缓冲读写的功能。不过，鉴于缓冲是一种简单而有效的策略，应用程序也可以自己在进程空间中安排缓冲区，来把多次操作合并成一次操作。事实上，C标准库中的标准IO函数，就负责在读写过程中管理进程空间的缓冲区。IPC和网络通信也经常用到相似的缓冲策略，以提高通信效率。

本章综合叙述了分级存储策略。计算机是一个多组件合作的整体，这些组件在性能上各有千秋，我们必须采用一定的方法来让它们合作，扬长避短，让各个组件发挥最大效率。在最近流行的AI集群和超级云平台上，经常能看到缓存和缓冲的应用，可见操作系统的经典设计历久弥新。

第32章 遍阅网络协议

前面的章节专注于计算机的内部，从这一章起转向计算机的外部，即网络功能。互联网的诞生晚于计算机，但它的发展极为迅速。通信协议模块，已经成为计算机操作系统密不可分的一部分。本章介绍网络协议的基础知识。

32.1 通信与互联网协议

通信是一件奇妙的事情，它让信息在不同的个体间传递。动物散发着特殊的气味，传递着求偶信息。人则说着甜言蜜语，向情人表达爱意。猎人吹着口哨，悄悄地围拢猎物。服务生则大声向后厨吆喝，要加两份炸鸡和啤酒。红绿灯指挥着交通，电视上播放着广告，法老的金字塔刻着禁止进入的诅咒。有了通信，每个人都和周围的世界进行着信息连接。

在通信这个神秘的过程中，参与通信的个体总要遵守特定的**协议**（Protocol）。在日常交谈中，我们无形中使用约定俗成的语法。两个人使用不同的语法，就是以不同的协议来交流，彼此会不知所云。像语言、语法这样的通信协议有特定的历史渊源，很难轻易改变，但人们还能自行创造通信协议。古人在长城上放狼烟，用来警告后方有外敌入侵。这样的警告之所以能成功传递，是因为人们已经约定狼烟代表了敌人入侵。狼烟代表了敌人入侵就是一个简单的通信协议。

协议可以更复杂。电报使用莫尔斯码通信。莫尔斯码用短按和长按的组合，来代表不同的英文字母。求救信号SOS，用莫尔斯码表示就是：

短短短 长长长 短短短

短表示短按，长表示长按。在莫尔斯码规定的协议中，连续的三个短信号代表S，三个长信号代表O。人们知道SOS是求助信息，是因

为我们还有个SOS代表求救的协议存在于脑海里。因此莫尔斯码的求救通信，依赖了一个两层协议组成的分层通信系统。

计算机之间的通信也是在不同的计算机个体之间传递信息。因为早期的计算机主要用作运算工具，所以不存在太强烈的数据需求。计算机放在一个机房内，借用缆线与各种外设连接起来。后来，计算机用户越来越多，用户之间分享数据的需求也越来越多。沿着电报、电话、电视已经积累的经验，计算机开始用电压、光照、电磁波等易于长距离传递的信号通信。

当然，计算机的通信不是那么简单。互联网是一个容许多方参与的开放网络。为了让不同设备沟通，通信协议必须标准化。计算机通信的内容又很丰富，可能是少量的文字，也可能是海量的音乐和电影。因此，通信协议又必须足够灵活，能包容不同的数据内容。在银行、医院、战场这样的关键场景中，计算机还必须保证通信的准确性和安全性。最后，参与通信的计算机很多。通信协议必须有一定的统筹策略，在保证网络通畅的前提下，尽快传递信息。计算机通过一套多层次的协议体系来满足上述的多方位需求。

32.2 协议分层

互联网通信协议以TCP/IP协议为核心，并通过多种多样的协议形式向上下游延伸。本节从底层协议开始简要介绍计算机协议。

1.物理层

计算机的基础协议在通信的物理介质。所谓的**物理层**（Physical Layer），是指光纤、电缆或者电磁波等真实存在的物理媒介。这些媒介可以传送物理信号，比如亮度、电压、振幅。计算机底层的信息是二进制码，用0和1构成的序列就可以代表信息，因此在物理层只需要约定两种物理信号来分别表示0和1即可，比如用高电压表示1，低电压表示0。从物理信号到二进制的约定就构成了物理层协议。针对特定的媒介，电脑可以有相应的接口，用来接收物理信号。随后计算机将利用相应的物理层协议，把物理信号解读成二进制序列。

2.连接层

连续的二进制序列就像没有标点的文言文一样让人头脑发昏。在**连接层**（Link Layer），我们把二进制序列分割成**帧**（Frame）。所谓的帧，是一段有限的二进制序列。连接层协议能帮助计算机识别二进制序列中所包含的帧。它规定特殊的0/1组合来作为帧的起始和结束。连接层协议还规定了帧的格式。帧中包含有**收信地址**（SRC，Source）和**送信地址**（DST，Destination），还有能够探测错误的**校验序列**（Frame Check Sequence）。当然，帧中最重要的是所要传输的数据。帧就像是一个信封，把数据包裹起来。

以太网（Ethernet）和Wi-Fi是现在最常见的连接层协议，分别用于有线网络和无线网络。树莓派的网口通信用的是以太网协议，而无线Wi-Fi用的自然就是Wi-Fi协议。因此，树莓派至少有两种方式接入网络。相应的，树莓派上也有两个**网络接口控制器**（NIC，Network Interface Controller），也就是所谓的“网卡”。这两个网卡分别使用以太协议和Wi-Fi协议进行通信。

通过连接层协议，我们可以指定帧的收信地址，从而把信息传递给其他的计算机设备。但遗憾的是，帧的收信地址只能是本地局域网内的。因此，连接层更像是一个社区的邮差，他认识社区中的每一户人家。社区中的每个人都可以将一封信，也就是一帧交给他。邮差把信送给同一社区的另一户人家。更远距离的通信还需要在更高的网络层实现。

3.网络层

网络层（Network Layer）的目的是让不同的社区之间通信。比如，让Wi-Fi局域网上的一台计算机和以太局域网上的另一台计算机通信，就需要一个“中间人”。这个“中间人”必须有以下功能。

- （1）能从物理层上为两个网络接收和发送0/1序列。
- （2）能同时理解两种网络的帧格式。

路由器（Router）就是为此而产生的“中间人”设备。一个路由器有多个网，因此路由器可以同时接入多个网络，并理解相应的连接层协议。在帧经过路由到达另一个网络的时候，路由会读取帧的信息，并改写以发送到另一个网络。所以路由器就像是在两个社区都有分支的邮局。一个社区的邮差将信送到本社区的邮局分支，而邮局会通过

自己在另一个社区的分支将信转交给另一个社区的邮差手中，并由另一个社区的邮差送到目的地。由于树莓派上有多个网卡，它也可以充当一个路由器。

我们说过，连接层的帧中只能记录本地的送信地址，如“第一条街第三座房子”或者“中心十字路口拐角的小房子”这样一些本地人才知道的地址描述。邮局收到这样的信件就傻眼了，这是要送到纽约还是东京。邮局只能抱着侥幸心理读一下信——也就是帧的数据部分。邮局发现，送信人居然也懂IP协议，在信的开头写上标准的邮编——IP地址。如果目的地社区也归这个邮局管，那么邮局工作人员就把信重新装到一个新的信封中，写上对应的本地地址，交给那个社区的邮差。然而，IP地址也可能不在邮局的管辖范围内。邮局会把信件转交到其他邮局。有时候一封信要通过多个邮局转交，才能最终到达目的地，这个过程叫作**路由**（Route）。邮局将分离的局域网络连接成了覆盖全球的互联网。

4.传输层

上面的三层协议让不同的计算机之间可以通信。但计算机中实际上有多个运行着的程序，也就是所谓的进程。每个进程都可能通信需求。这就好像一所房子里住了好几个人。如何让信准确送到某个人手里呢？遵照与之前相同的逻辑，在网络层协议的数据部分增加**传输层**（Transport Layer）信息。我在信纸上增加新的信息，也就是收信人的姓名。大楼管理员从邮差手中接过信，会根据收信人，将信送给房子中的某个人。

传输层协议，比如TCP协议和UDP协议，都使用**端口号**（Port Number）来识别收信人。在写信的时候，我们写上目的地的端口。当信到达目的地的管理员手中后，他会根据传输层协议，识别端口号，将信送给不同的人。TCP和UDP协议是两种不同的传输层协议。UDP协议类似于信件交流过程，一封信包含所有的信息。TCP协议则好像两个情人间的频繁通话。他们要表达的感情太多，以至于连续的发信。另一方必须将这些信按顺序排列起来，才能看明白全部的意思。此外，TCP协议还能控制网络交通，避免网络阻塞。由于完善的功能，TCP也成了最常用的传输层协议。

5.应用层

通过上面的几层协议，我们已经可以在任意两个进程之间进行通信。然而，在**应用层**（Application Layer）上，还可以有更高层的协议。不同类型的进程就像从事不同行业的人。有的是律师，有的是外交官。某些行业会有特定的职业用语规范。律师之间的通信会用严格的律师术语，以免产生纠纷。外交官之间的通信，也要符合一定的外交格式，以免发生外交事故。如果是情报机构，则要通过暗号来加密信息。同样，某个类型的应用可以使用某个应用层协议，进一步规范用语。

应用层的协议包括用于Web的HTTP协议，这是浏览器工作的基础。DNS协议也是应用层协议，从而允许我们使用如douban.com这样的域名。应用层的IMAP协议用于E-mail传输。还有些加密协议让数据能安全地传递。应用层协议不但让通信更稳定也更完善，还让计算机能提供更丰富的互联网服务。

计算机通信从物理信号出发，最终实现了复杂的互联网通信，靠的就是网络协议。这些高层协议像邮差、邮局、大楼管理员一样，传递符合特定用语规范的信息。通过不同层次的封装，我们不再关心底层协议，专注于高层信息的编辑和发送。这些看起来繁杂得像森林的网络协议，是互联网最重要，也最常被忽视的基础设施。

第33章 树莓派网络诊断

通过对网络协议的介绍，我们已经了解了互联网通信的基本原理。互联网让树莓派变得更加强大。但这也意味着，网络问题会让人非常恼火。下面介绍树莓派常用的网络诊断命令，它们能帮助我们发现网络问题。

33.1 基础工具

网络诊断的第一步是了解自己的设备，比如有哪些接口，IP地址都是什么。使用下面的命令来显示**网络接口**（Interface）信息，如接口名称、接口类型、接口的IP地址、硬件的MAC地址等。

```
$sudo ip address show
```

ARP协议用在局域网内部。借用ARP协议设备可以知道同一局域网内的IP-MAC对应关系。当访问一个本地IP地址时，设备根据该对应关系，与对应的MAC地址通信。通过ARP工具，可以知道局域网内的通信是否正常。

```
$sudo arp -a
```

显示本地存储的IP地址和MAC地址的对应关系。

安装arping工具：

```
$sudo apt-get install arping
```

然后使用命令：

```
$sudo arping -I eth0 192.168.1.1
```

经eth0接口，发送ARP请求，查询IP为192.168.1.1设备的MAC地址。

安装arp-scan工具：

```
$sudo apt-get install arp-scan
```

然后使用下面的命令查询整个局域网内所有IP地址的对应MAC地址：

```
$sudo arp-scan -l
```

安装tcpdump工具：

```
sudo apt-get install tcpdump
```

使用命令：

```
$sudo tcpdump -i en0 arp
```

监听en0接口的ARP协议通信。

33.2 网络层

网络层是一个广域的互联网，互联网上的设备用IP地址识别。ping命令是向某个IP地址发送ICMP协议的ECHO_REQUEST请求。收到该请求的设备将返回ICMP回复。如果ping请求到某个IP地址，则说明该IP地址的设备可以经网络层顺利到达。

```
$ping 192.168.1.1
```

向IP地址192.168.1.255发送ICMP请求。如果该地址的ICMP没有被禁用，那么在该网上的设备将回复：

```
$ping 192.168.1.255
```

向广播地址192.168.1.255发送ICMP请求。如果ICMP没有被禁用，那么在该网上的设备将回复。

```
PING 192.168.1.255 (192.168.1.255): 56 data bytes
64 bytes from 192.168.1.255: icmp_seq=0 ttl=64 time=196.613 ms
64 bytes from 192.168.1.255: icmp_seq=1 ttl=64 time=133.379 ms
```

需要注意的是，许多网络设备会禁用ICMP。即使ping请求不到一个设备，并不一定是网络层故障，ping的结果只能作为参考。

如果两个设备有相同的IP地址，将导致IP冲突。许多网络是由DHCP协议自动分配IP地址的，这样可以极大减少IP冲突的可能性。DHCP服务器与设备达成协议，设备将在一定时间内占据某个IP地址，而DHCP服务器不再把该IP地址分配给别人。

```
$sudo dhclient -v -r
```

更新DHCP租约，设备将释放IP地址，再从DHCP服务器重新获得IP地址。

```
$sudo ifconfig wlan0 192.168.1.106 up
```

将接口wlan0的IP地址设置成192.168.1.106。

```
$sudo nano /etc/dhcpd.conf
```

编辑 */etc/dhcpd.conf* 文件，在文件末尾加入：

```
interface eth0
static ip_address=192.168.1.106
```

可将接口eth0的默认IP地址设置成192.168.1.106。

33.3 路由

局域网通过路由器接入广域的互联网。互联网上的通信往往要经过多个路由器接力。途中路由器的故障，可能导致互联网访问异常。

```
$netstat -nr
```

显示路由表。从路由表中，可以找到网关。网关是通向更加广域网络的出口。

```
$traceroute 74.125.128.99
```

追踪到达IP目的地的全程路由。

```
$sudo traceroute -I 74.125.128.99
```

通过ICMP协议追踪路由。ICMP协议经常会被禁用，所以会返回“*”的字符串。通过TCP协议，经80端口追踪路由，TCP协议的默认端口80很少会被禁用。

```
$sudo traceroute -T -p 80 74.125.128.99
```

33.4 网络监听

在Linux下，tcpdump是一款网络抓包工具。它可以监听网络接口不同层的通信，并过滤出特定的内容，比如特定协议、特定端口等。我们已经使用tcpdump监听了ARP协议通信，下面介绍更多的监听方式。

- 监听en0接口的所有通信。

```
$sudo tcpdump -i en0
```

- 用ASCII显示en0接口的通信内容。


```
$sudo tcpdump -A -i en0
```

- 显示en0接口的8080端口的通信。

```
$sudo tcpdump -i en0 'port 8080'
```

- 显示eth1接口来自192.168.1.200的通信。

```
$sudo tcpdump -i eth1 src 192.168.1.200
```

- 显示eth1接口80端口、目的地为192.168.1.101的通信。

```
$sudo tcpdump -i eth1 dst 192.168.1.101 and port 80
```

- 将lo0接口的通信存入文件 **record.pcap**，方便阅读。

```
$sudo tcpdump -w record.pcap -i lo0
```

通过tcpdump能知道不同协议层传输的内容，进而诊断网络问题的原因。

33.5 域名解析

DNS在域名和IP之间进行翻译，DNS故障会导致用户无法通过域名访问某个网址。

```
$host www.sina.com.cn
```

DNS域名解析，返回域名对应的IP地址。你可以通过这个域名来检查计算机是否能正确进行域名解析。

本章对网络诊断相关命令的介绍很简略，只能给你留下一个粗浅的印象。毕竟，Linux下的网络命令非常庞杂，相关介绍足以构成一本书。你也可以通过上面各个命令的文档来详细了解它们的用法。

第5部分 树莓派小应用

第5部分将会介绍一些基于树莓派开发的小应用。开发这些小应用，不仅需要了解树莓派，还需要运用很多Linux、计算机网络、程序编写相关的知识。树莓派可以做的事情远远不止本部分所介绍的这些，衷心希望读者朋友们在阅读第5部分后有所启发，用树莓派做出更有意思的发明创造。

第34章 树莓派平板电脑

平板电脑对于我们并不是一个新鲜的概念。早在1989年世界上第一台平板电脑GRiDPad就已经问世了。它的设计和制造者，是来自英国的GRiD公司，这家公司目前依然存在，主要生产军用电子产品。

34.1 平板电脑

GRiDPad平板电脑重约2kg，拥有1兆字节内存，液晶屏可以显示24行，每行80个英文字符。它运行着当时世界上最先进的MS-DOS 3.3操作系统。由于制造成本昂贵、使用不便等原因，这款平板电脑的出现并没有引发平板电脑的流行。

目前市面上的平板电脑分属几家厂商，运行着不同的操作系统。苹果公司的iPad运行着iOS操作系统，亚马逊的Kindle Fire运行着Android操作系统，微软的Surface运行着Windows操作系统。这些平板电脑一般都配备一个支持多点触控的液晶触摸显示屏，有的平板电脑还配有触控笔。用户可以在平板电脑上安装应用软件、输入文字、连接互联网。

如今，树莓派3B型已经具备1吉字节的内存，超过GRiDPad的一千倍，与市面上很多低端平板电脑配置相当。树莓派官方也推出了一款7英寸的触摸显示屏，这里我们将尝试如何用这款触摸显示屏和树莓派一起，制作一个可以满足日常使用的平板电脑。

34.2 硬件介绍

做一台平板电脑需要涉及一些硬件和软件。硬件方面，我们用树莓派作为平板电脑的核心处理部件，官方7英寸触摸屏作为主要输入/输出设备。树莓派官方的7英寸触摸屏拥有800×480的分辨率，配备一

块拓展电路板，集成了供电和信号转换功能，让这块屏幕与树莓派的连接变得尤为简单，只需要连接GPIO的电源与一组排线即可。软件方面，常见的平板操作系统有安卓和iOS。值得一提的是，Raspbian自带了一般Linux发行版没有的配合官方显示屏的10点触摸的驱动，这样触摸功能和文字输入就解决了^[1]。

我们还要考虑输入输出设备。虽然最终的平板电脑不需要使用键盘鼠标来操作，但为了在初始状态下更方便地配置树莓派，我们还需要一套有USB接口的键盘和鼠标。市面上还有很多与这款显示器配套的外壳。它可以保护我们的树莓派和屏幕拓展板，想把自制平板电脑带在路上使用的朋友们不妨考虑购买。

34.3 硬件的安装

树莓派官方显示屏的安装非常简单。我们需要连接的部件有树莓派、触摸屏、触摸屏拓展板（这是与触摸屏一起出售的一块小电路板）。

首先，连接拓展板与触摸屏。如果你购买的树莓派显示屏已经和拓展板连接好了，那么请直接跳过此步骤。如果没有，那么请参照图34-1将触摸屏上的橙色排线插进拓展板对应的位置。

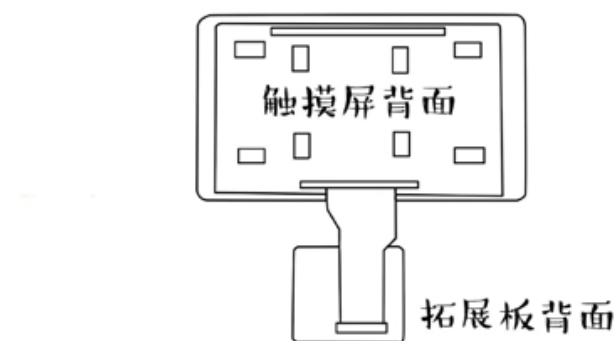


图34-1 连接显示屏和拓展板

然后，连接拓展板与树莓派之间的DSI排线，如图34-2所示。

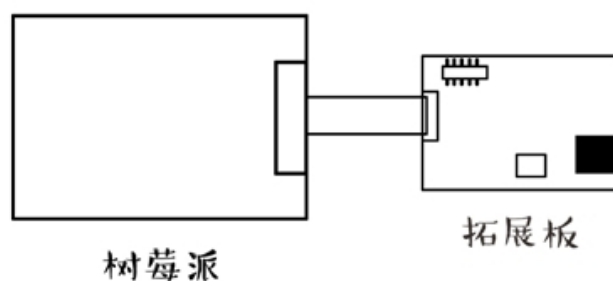


图34-2 连接树莓派和拓展板之间的DSI排线

最后，使用两根导线连接树莓派和拓展板的电源接口。两个电源接口分别是5V电压和GND（地线），如图34-3所示。

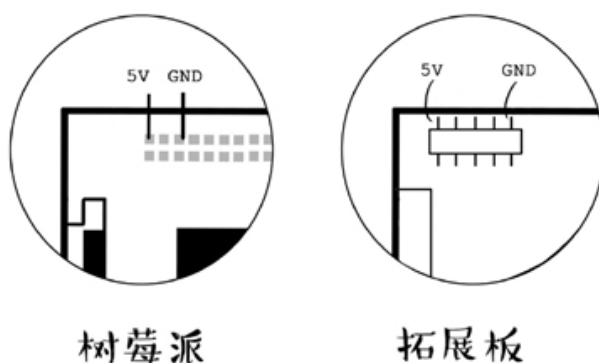


图34-3 连接树莓派和拓展板的导线

此外，还可以用显示屏自带的5个螺丝和金属柱将树莓派和拓展板固定起来，这样使用起来更方便。完成上述步骤，跟往常一样将树莓派和电源连接。这样树莓派平板电脑的硬件部分就完成了。需要注意的是，树莓派官方的触摸显示屏是使用树莓派本身供电的。如果电源功率不足，则很可能在使用的时候出现屏幕抖动、花屏等情况，所以使用质量好的USB电源很重要。如果一切正常，那么显示屏上会出现树莓派的Raspbian OS操作系统界面。

34.4 配置操作系统

因为我们还没有设置好树莓派的软件系统，所以需要将USB鼠标和键盘连接在树莓派上完成最初的软件配制。

1. 设置屏幕方向（可选）

开机后，如果屏幕显示方向是倒立的，那么可以通过配置启动文件来解决这个问题。

树莓派的启动控制文件的路径是 `/boot/config.txt`。因为这个文件只有Linux系统管理员（root）用户才可以编辑，所以用命令行来编辑更加简单。将键盘鼠标通过USB连接树莓派，通过桌面顶部的菜单 Menu → Accessories → Terminal 打开树莓派的命令行，然后输入以下命令：

```
$sudo nano /boot.config.txt
```

这样就可以用管理员账号使用nano编辑器打开配置文件了。在前面的章节中，已经介绍了nano编辑器的基本操作方法，也可以使用命令行编辑器打开。想要改变显示屏的方向，只需要在该文件的底部加入这样一行：

```
lcd_rotate=2
```

这一行会让屏幕显示180°旋转，这样就可以正常使用树莓派平板电脑了。

2.安装虚拟键盘

虚拟键盘是平板电脑不可缺少的一个软件。首先，把树莓派操作系统里的软件升级到最新的稳定版。用上面的方法打开Terminal，依次执行以下指令。

```
$sudo apt-get update  
$sudo apt-get upgrade
```

然后，安装虚拟键盘程序。

```
$sudo apt-get install matchbox-keyboard
```

最后，用菜单重启树莓派，激活虚拟键盘。打开虚拟键盘，先单击树莓派图标，选择Accessories-> Keyboard。打开虚拟键盘后，桌面将如图34-4所示。

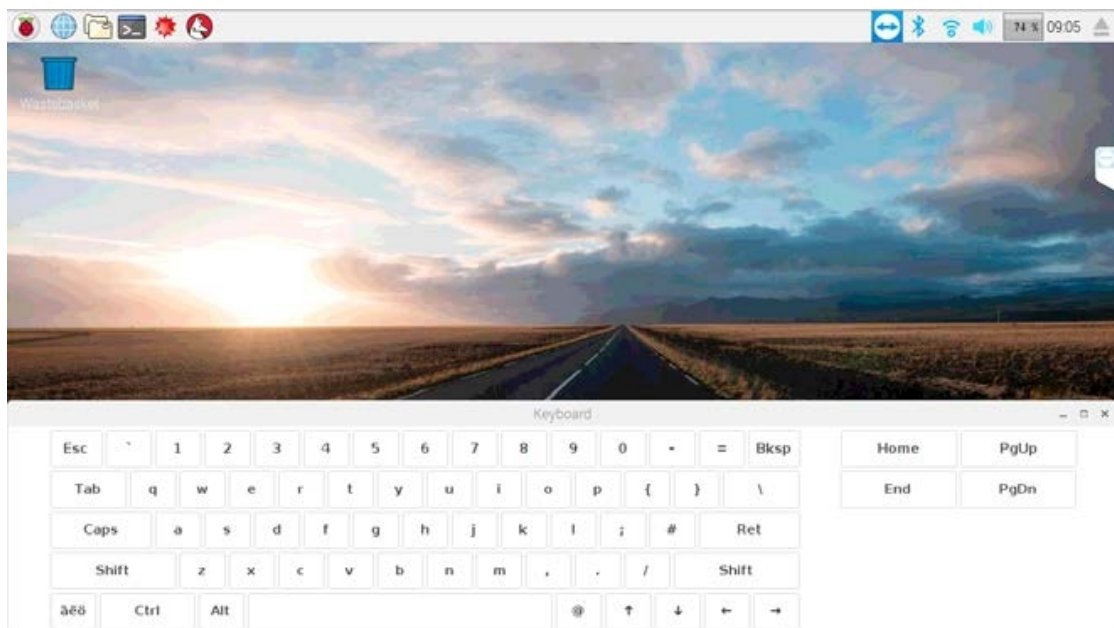


图34-4 软键盘

如果在Accessories中找不到Keyboard选项，可以单击树莓派图标，选择 Preferences->Main Menu Editor。在这个对话框中选中 Accessories分类里面的Keyboard选项，这样Keyboard选项就可以在树莓派菜单中找到了。

有了触摸屏和软键盘，我们就可以顺畅地在图形化界面用触屏的方式来使用树莓派。此外，我们已经介绍过树莓派下的图形化应用软件。这些应用软件可以满足我们日常的需求。

[1] 在树莓派的官网可以找到官方显示屏的授权零售商。在淘宝等国内电商平台上也可以找到这款显示器，委托在外国的朋友们代购也可以。

第35章 天气助手

不知道读者们有没有出门前查看天气预报的习惯。如果你和我一样总是忘记，就难免在下雨天淋雨了。本章将会通过树莓派给自己定时发送天气提醒。

35.1 读取互联网API

1. 选择天气预报服务

API是Application Programming Interface（应用编程接口）的英文缩写，是电脑程序之间交互信息的方式。本节将使用互联网API来获取天气预报信息。

网上有很多提供天气预报API的服务对于个人用户都是免费的。本节我们将以和风天气^[1]作为例子。

在开始之前，先到和风天气网站注册一个账户。注册完成后，登录产品控制台可以看到自己的API接口信息，如图35-1所示。注意，个人认证Key与密码一样是私密信息，不可以公开放在网上。

2. 测试API

和风天气的API使用起来非常简单，我们甚至可以直接使用浏览器来获取API结果。不过这里，我要推荐一款名叫Postman的测试工具，它用于测试符合HTTP协议的API。Postman是一个Chrome浏览器拓展，可以从Google Chrome的官方商店中下载。Postman的界面，如图35-2所示。



图35-1 API接口

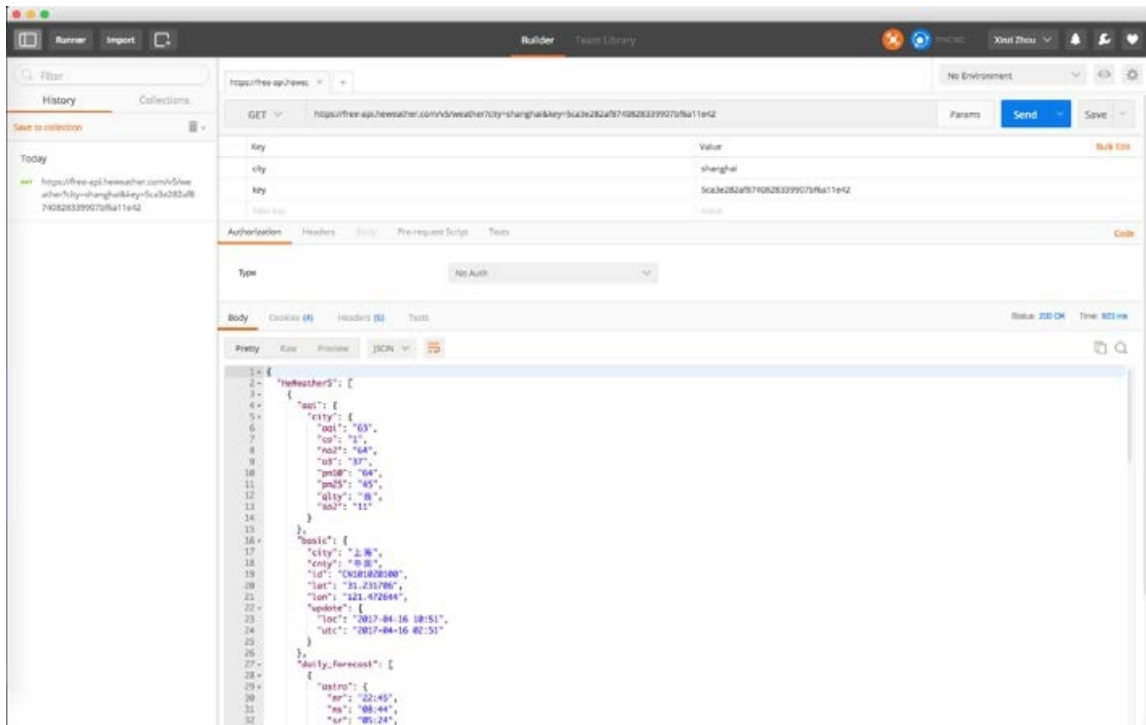


图35-2 Postman的界面

我们要使用的API是“全部天气”^[2]，在Postman中新增一个标签页。首先，在标签页最上面左侧的下拉列表中选择GET，因为我们要使用的API是一个HTTPGET请求。然后，在下拉列表右侧写着“Enter request URL”的输入框中输入API网址，即https://freeapi.heweather.com/v5/weather。它被称作API的端点（Endpoint）。单击网址右边的“Params（参数）”按钮，你会看到网址下面出现了一个键值列表。在列表中添加下面两项，如表35-1所示。

表35-1 键值列表

Key（键）	Value（值）
city	城市拼音，例如 shanghai
key	和风天气用户面板上的个人认证 Key

单击蓝色的“Send”按钮，稍等片刻，我们就会在Body区域看到服务器的返回，如图35-3示。

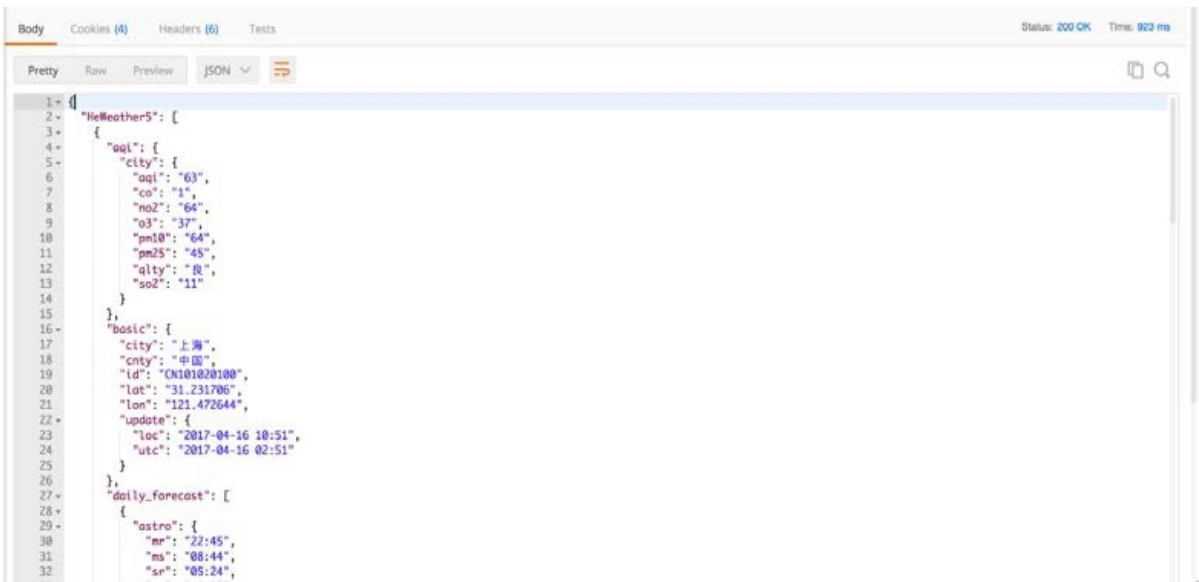


图35-3 服务器的返回

服务器的返回是一个JSON字符串，Postman会自动将它格式化，显示成分行缩进的样式。JSON是一种通用的网络信息传输格式。简单介绍这种返回格式。返回的JSON字段，所有重要信息都在一个HeWeather5的值内。HeWeather5是一个数组，每个数组元素是一个城市的天气信息。通常情况下，这个数组只会返回一个元素。每个天气信息中有这个城市的

天气预报数据。其中的suggestion段，是给人们的生活提示。我们接下来的提醒功能将会使用这个字段的数据，如图35-4所示。

```
"suggestion": {
  "air": {
    "brf": "中",
    "txt": "气象条件对空气污染物稀释、扩散和清除无明显影响，易感人群应当减少室外活动时间。"
  },
  "comf": {
    "brf": "较舒适",
    "txt": "白天有雨，从而使空气湿度加大，会使人们感觉有点儿闷热，但早晚的天气很凉爽、舒适。"
  },
  "cw": {
    "brf": "不宜",
    "txt": "不宜洗车，未来24小时内有雨，如果在此期间洗车，雨水和路上的泥水可能会再次弄脏您的爱车。"
  },
  "drsg": {
    "brf": "舒适",
    "txt": "建议着长袖T恤、衬衫加单裤等服装。年老体弱者宜着针织长袖衬衫、马甲和长裤。"
  },
  "flu": {
    "brf": "易发",
    "txt": "相对于今天将会出现大幅度降温，空气湿度较大，易发生感冒，请注意适当增加衣服。"
  },
  "sport": {
    "brf": "较不宜",
    "txt": "有降水，推荐您在室内进行健身休闲运动；若坚持户外运动，须注意携带雨具并注意避雨防滑。"
  },
  "trav": {
    "brf": "适宜",
    "txt": "有降水，温度适宜，在细雨中游玩别有一番情调，可不要错过机会呦！但记得出门要携带雨具。"
  },
  "uv": {
    "brf": "弱",
    "txt": "紫外线强度较弱，建议出门前涂擦SPF在12-15之间、PA+的防晒护肤品。"
  }
}
```

图35-4 数据字段

3.在树莓派中使用网络API

bash是树莓派中最常用的脚本语言之一。我们将使用bash来编写程序，从天气预报的API中读取信息。

我们需要使用curl工具来调用远程API，再使用jq工具来解析返回的天气信息。

curl工具是树莓派中自带的，而jq工具需要安装：

```
$apt-getinstall jq
```

下面是一个获取天气预报信息的代码。读者可以将这段代码输入一个名叫 ***call_weather_api.sh*** 的文件中：

```
#!/usr/bin/env bash

CITY=shanghai
TOKEN=5ca3e282af8740828339907bf6a11e42

WEATHER=$(curl "https://free-
api.heweather.com/v5/weather?city=${CITY}&key=${TOKEN}")

SUGGESTIONS=$(echo ${WEATHER} | jq -r '.HeWeather5[0].suggestion |
values[].txt')

echo ${SUGGESTIONS}
```

在上面这段代码中，你需要把变量city换成你所在的城市的英文名，变量key换成你从和风天气网站申请到的Key。和风天气支持的城市列表可以从网站的API说明文档中找到^[3]。

这段代码把天气预报的信息文本最后储存在了一个叫SUGGESTIONS的变量中，并显示在了屏幕上。运行这段代码的方式和运行其他bash脚本一样，只需要在命令行Terminal中输入下面的语句：

```
$bash call_weather_api.sh
```

其运行结果如下：

```
气象条件有利于空气污染物稀释、扩散和清除，可在室外正常活动。
白天天气晴好，但烈日炎炎您会感到很热，很不舒适。
属中等强度紫外线辐射天气，外出时建议涂擦 SPF 高于 15、PA+的防晒护肤品，戴帽子、太阳镜。
```

显然，这段代码的运行结果取决于设置的城市和当时天气预报的情况，所以读者朋友都会得到不同的运行结果，这是正常的。运行脚本时遇到异常可以阅读Python编程的相关知识来检查错误。

这段代码只是简单地提取了API返回的建议文本，其实和风天气的API返回的内容还有很多，读者可以自己尝试增加更多内容。

35.2 发送邮件

获得了天气预报信息，下一步就要把这个信息发送到手机上了。把信息从树莓派推送到手机的方法有很多。这里介绍发送邮件的方式。

电子邮件是一个标准的互联网协议。使用电子邮件发送天气提醒的好处主要有两方面：一方面，简单免费，现在只要计算机连接互联网，就可以使用免费的邮箱服务来发送邮件；另一方面，邮件的内容可以很丰富，甚至可以包含多媒体信息。

下面的代码可以实现发送邮件的功能。复制刚才创建的文件 *call_weather_api.sh* 到 *send_email.sh*，将下面的代码输入新文件下方。

```
SERVER="smtp.gmail.com:587"
FROM="imdreamrunner@gmail.com"
TO="imdreamrunner@gmail.com"
SUBJECT="天气预报 $(date)"
MESSAGE="${SUGGESTIONS}"
CHARSET="utf-8"
USERNAME="imdreamrunner@gmail.com"
PASSWORD="邮箱密码"

sendmail \
    -f ${FROM} \
    -t ${TO} \
    -u ${SUBJECT} \
    -s ${SERVER} \
    -m ${MESSAGE} \
    -xu ${USERNAME} \
    -xp ${PASSWORD} \
    -v -o message-charset=${CHARSET}
```

有的电子邮箱服务商对于是使用程序发送邮件有额外的安全要求。如果发送邮件失败，那么这段程序会打印出失败原因，里面可能会有服务商要求的安全设置方法。

如果邮件发送成功，那么你将会收到一封电子邮件，如图35-5所示。



图35-5 邮件截图

下面用计划任务的方式，让树莓派在特定时间发出提醒邮件。我们可以用之前介绍过的cron，输入命令：

```
$crontab -e
```

进入编辑页面。如果需要每天8点半发送邮件，那么增加：

```
30 8 * * * bash /path/to/send_mail.sh
```

需要注意的是，这里的/path/to要替换成 *send_mail.sh* 所在的路径。经过这样的修改，每天早上8点半cron就会启动 *send_mail.sh*。这个bash脚本工作后会发送邮件告诉我们今日的天气。

[1] 和风天气www.heweather.com。

[2] API的详细介绍可以在[HTTP://www.heweather.com/documents/api/v5/weather](http://www.heweather.com/documents/api/v5/weather)中找到。

[3] 具体网址是www.heweather.com/documents/city。

第36章 架设博客

博客是网络日志的中文音译。只需要去公共博客网站，例如Blogger上创建一个账号即可拥有博客了。本章将要介绍一件听起来很酷的事情——把博客架设在树莓派上。自己架设的博客成本更低，只需要付树莓派的电费和宽带费，而且更加自由。本章会使用一个国人开发的开源博客系统Typecho^[1]。

36.1 安装服务器软件

Typecho使用的编程技术是PHP。在开始创作之前，需要在树莓派上安装可以运行PHP程序的必要软件。

第一步，更新树莓派本地软件库版本。

```
$sudo apt-get update && sudo apt-get upgrade
```

第二步，安装Apache。Apache是一个网页的服务器。网站的访客需要通过Apache才能看到你的博客的内容。

```
$sudo apt-get install apache2 apache2-utils
```

第三步，安装PHP。PHP是一个脚本语言。因为Typecho程序是由PHP写的，所以要在树莓派上安装PHP才能正常使用。

```
$sudo apt-get install libapache2-mod-php5 php5 php5-curl php5-sqlite
```

第四步，安装数据库SQLite。Typecho会使用SQLite作为数据库。

```
$apt-get install sqlite3 php5-sqlite
```

此外，需要配置一下PHP5-sqlite模块。在命令行中使用nano或者其他文本编译器来编辑配置文件 `/etc/php5/cli/conf.d/20-sqlite3.ini`，脚本如

下：

```
$sudo nano /etc/php5/cli/conf.d/20-sqlite3.ini
```

至此，我们可以测试一下Apache和PHP是否正常工作。重启Apache服务器。

```
$apachectl restart
```

删除 ***/var/www/html/index.html***，并创建文件 ***/var/www/html/index.php***。这两个操作的命令为：


```
$rm /var/www/html/index.html  
$touch /var/www/html/index.php
```

在新创建的文件 ***/var/www/html/index.php*** 中输入下面的内容：

```
<?php phpinfo(); ?>
```

此时在树莓派中打开<http://localhost/>，如果网页能正常显示，则说明安装成功，如图36-1所示。

PHP Version 5.6.30-0+deb8u1



System	Linux raspberrypi 4.9.40-v7+ #1022 SMP Sun Jul 30 11:16:10 BST 2017 armv7l
Build Date	Apr 14 2017 15:27:54
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/apache2
Loaded Configuration File	/etc/php5/apache2/php.ini
Scan this dir for additional .ini files	/etc/php5/apache2/conf.d
Additional .ini files parsed	/etc/php5/apache2/conf.d/05-opcache.ini, /etc/php5/apache2/conf.d/10-pdo.ini, /etc/php5/apache2/conf.d/20-curl.ini, /etc/php5/apache2/conf.d/20-json.ini, /etc/php5/apache2/conf.d/20-pdo_sqlite.ini, /etc/php5/apache2/conf.d/20-readline.ini, /etc/php5/apache2/conf.d/20-sqlite3.ini
PHP API	20131106
PHP Extension	20131226
Zend Extension	220131226
Zend Extension Build	API220131226,NTS
PHP Extension Build	API20131226,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
IPv6 Support	enabled
DTrace Support	enabled
Registered PHP Streams	https, ftps, compress.zlib, compress.bzip2, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, tls, tlsv1.0, tlsv1.1, tlsv1.2
Registered Stream Filters	zlib.*, bzip2.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk

This program makes use of the Zend Scripting Language Engine:
Zend Engine v2.6.0, Copyright (c) 1998-2016 Zend Technologies
with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, by Zend Technologies




图36-1 测试成功页面

36.2 安装Typecho

下面安装Typecho。

第一步，切换到管理员账号。

```
$sudo su
$cd /var/www
```

第二步，下载源代码。我们可以从Typecho官网获取下载链接^[2]。

```
$wget https://github.com/typecho/typecho/releases/download/v1.0-14.10.10-release/1.0.14.10.10.-release.tar.gz
```

第三步，处理源代码。

```
$tarzxvf 1.0.14.10.10.-release.tar.gz
$rm -rf html
$mv build/ html
$chown -R www-data html
```

第四步，运行安装脚本。在浏览器中打开http://localhost/。根据提示进行一些初始化配置，就可以访问博客了。博客首页的地址是http://localhost/，而管理面板在http://localhost/admin/上，如图36-2所示。



图36-2 博客首页

36.3 让别人可以访问你的网站

现在私人云盘已经成功地运行在了本地的局域网内。如何将文件分享给朋友呢？这就需要让树莓派有一个固定的IP地址和指向这个地址的域名。不过，电信运营商一般不会向家庭用户提供固定IP地址，就算有，这项服务通常也十分昂贵。我们有两种方案可以解决这个问题，即使用动态DNS服务，或者使用内网穿透服务。

动态DNS服务其实早已出现。例如，“花生壳”就从2006年开始提供免费的动态DNS解析服务。不过，普通宽带用户的网络有时候不一定那么稳定，使用动态DNS服务的网站经常会面临无法访问的情况。这里，我们将介绍另一种服务，也就是内网穿透，这是一个更简单快捷的解决方案，可以用ngrok来实现。ngrok是一款非常好用的内网穿透软件和服务。也就是说，通过ngrok，别人可以访问你架设在树莓派上的网站。

首先，下载ngrok。打开ngrok的官方下载页面^[3]，并找到下载链接。然后在树莓派上下载这个文件。在命令行中输入下面的命令。

```
$wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-arm.zip
```

下载完成后，需要使用命令解压它。

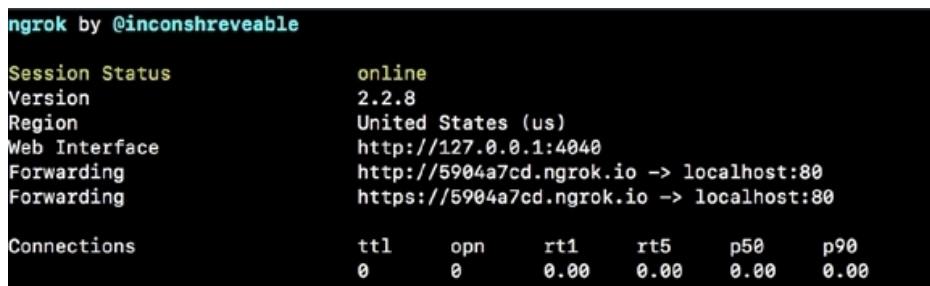
```
$unzip ngrok-stable-linux-arm.zip
```

这时，在当前目录就会出现一个名为 **ngrok** 的可执行文件，然后我们便可以启动ngrok了。使用命令：

```
$. /ngrok http 80
```

这个命令的意思是将会创建一个通道，让互联网上的用户可以访问本地80端口上的HTTP网站内容。

运行完后会看到如图36-3所示的界面。



```
ngrok by @inconshreveable

Session Status      online
Version             2.2.8
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://5904a7cd.ngrok.io -> localhost:80
Forwarding           https://5904a7cd.ngrok.io -> localhost:80

Connections
```

	ttl	opn	rt1	rt5	p50	p90
	0	0	0.00	0.00	0.00	0.00

图36-3 ngrok运行页面

界面中 Forwarding 一项出现的网址，也就是图 36-3 中的 `http://5904a7cd.ngrok.io`。这个地址就是ngrok创建的公网地址。在浏览器中试着访问这个地址，你会发现从这个网址访问到的网站和在树莓派中

使用<http://localhost/>访问到的网站是同一个。把这个网址发送给朋友，他们就可以访问你在树莓派中架设的博客了。

[1] 官网地址<http://typecho.org/>。

[2] 下载查询页面<http://typecho.org/download>。

[3] ngrok网址<https://ngrok.com/download>。

第37章 离线下载

互联网上的文件变得越来越大，于是离线下载的工具就诞生了，它可以在我们上学或上班的时候，帮我们下载大文件。树莓派是一个可以联网使用，可以外接移动硬盘而且耗电极低的电脑。我们可以用它下载需要的资源。

37.1 安装下载工具Aria2

我们用来做离线下载工具的软件叫Aria2。Aria2是一款轻量级的命令行下载工具，支持包括HTTP、BT在内常见的下载协议。Aria2的操作需要通过命令行来完成，但是我们之后会介绍安装一个网页版的图形化工具来控制Aria2。

Aria2没有发布在包管理工具上，所以没有办法用apt-get命令来安装。我们需要从源代码来安装。首先下载Aria2的源代码^[1]，可以用下面的命令实现：

```
$cd ~/Downloads
$wget https://github.com/aria2/aria2/releases/download/release-1.31.0/aria2-1.31.0.tar.gz
```

Aria2源代码的压缩包是.tar.gz文件格式。这是一种在Linux中常见的压缩文件格式。解压这个压缩文件可以用Linux中的tar工具，方法如下：

```
$tar zxvf aria2-1.31.0.tar.gz
```

tar命令后的zxvf是四个参数。z代表Gzip格式，x代表解压，v代表显示操作过程，f代表该操作对应一个文件。

编译Aria2程序。先切换到Aria2源代码所在的文件夹：

```
$cd aria2-1.31.0/
```

使用configure命令配置编译选项：

```
$. /configure
```

随后，使用make命令编译程序：

```
$make
```

使用make install命令安装程序到系统中：

```
$sudo make install
```

37.2 Aria2的使用

Aria2被安装好后可以使用命令行工具aria2c下载。一种使用方法是后续待下载文件的HTTP地址，例如：

```
$aria2c http://baidu.com/some-file.zip
```

另一种使用方法是通过BT下载文件，即说明BT文件的URL地址，例如：

```
$aria2c http://example.org/mylinux.torrent
```

还有使用磁力链接下载文件，即说明磁力链接地址，例如：

```
$aria2c 'magnet:?xt=urn:btih:248D0A1CD08284299DE78D5C1ED359BB46717D8C'
```

这里介绍了下载HTTP文件、BT文件和磁力链接的方法，只要保持树莓派的开机状态，就可以让树莓派一天24小时下载了。

37.3 远程使用Aria2

想要远程使用Aria2，就要有能访问树莓派的命令行。最简单快捷的方法是使用类似Teamviewer的远程桌面软件，使用它就可以远程控制命

命令行窗口进行下载，可以随意中断远程桌面连接，不需要额外操作树莓派就会自动继续下载。

如果不希望使用Teamviewer之类的软件，那么可以使用ssh命令来访问树莓派。第9章已经介绍过如何使用SSH连接树莓派了。这里介绍使用screen命令在后台执行下载任务。先通过apt-get安装screen：

```
$sudoapt-getinstall screen
```

在执行下载命令之前，使用screen命令新建一个会话：

```
$screen
```

在新建的会话中，输入下载命令开始下载，例如：

```
$aria2c http://baidu.com/some-file.zip
```

假如要下载的文件很大，那么在文件正在被下载的过程中，依次按下键盘上的快捷键Ctrl+A+D，便可将当前会话卸载，此时之前的下载操作会从命令行窗口中消失。

此时结束SSH连接并不会中断正在进行的下载。如果需要查看正在进行的下载任务，那么可以使用screen-r命令重新安装之前被卸载的会话。

```
$screen -r
```

37.4 安装图形化下载管理工具

读者可能会觉得用命令行操作Aria2不够直观方便，下面介绍一个图形化的下载管理工具WebUI。webui-aria2是一个用来管理Aria2的图形化工具。

首先准备webui-aria2的代码。这里的操作和之前下载Aria2的类似。不过这次我们不是使用wget命令通过HTTP下载源代码的压缩包，而是使用git命令直接下载GitHub资料库，步骤如下：

```
$cd ~/Downloads
$git clone git@github.com:ziahmza/webui-aria2.git
$cd webui-aria2/
```

然后将Aria2在后台运行。如果通过图形化界面控制树莓派，那么非常简单，只需要打开一个新的Terminal窗口，输入下面的命令，并保持这个窗口不被关闭即可。

```
$aria2c --enable-rpc --rpc-listen-all
```

如果使用纯命令行界面，那么可以使用screen，运行screen新建一个会话。

```
$screen
```

输入aria2c命令，使Aria2在这个会话中保持执行。最后在键盘上按快捷键Ctrl+A+D卸载当前会话。

运行WebUI。运行WebUI需要用到node.js，node.js的安装方法也很简单。下面两行代码是node.js官方提供的在Linux下node.js 6.x版的安装方法。直接在命令行中输入并执行下面两句代码即可。

```
$curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
$sudoapt-getinstall -y nodejs
```

第一句代码首先用curl工具下载了一段bash脚本，并使用bash执行这段脚本。这段脚本给apt-get添加了一个软件源，而在这个软件源中可以下载到node.js 6.x。第二句代码是使用apt-get命令安装nodejs。

用上面的脚本安装好node.js后，就可以用它来运行WebUI服务器了。

```
$node node-server.js
```

终端中会输出一行类似“WebUIAria2Server is running on http://localhost:8888”的文字，这行文字表明WebUI已经在8888端口运行。现在就可以用浏览器访问WebUI了。

如果是在树莓派本地访问，那么可以直接访问<http://localhost:8888>；如果是在别的电脑上访问，那么可以访问<http://<hostname>:8888>，其中hostname是树莓派在局域网中的主机名。WebUI的界面如图37-1所示。

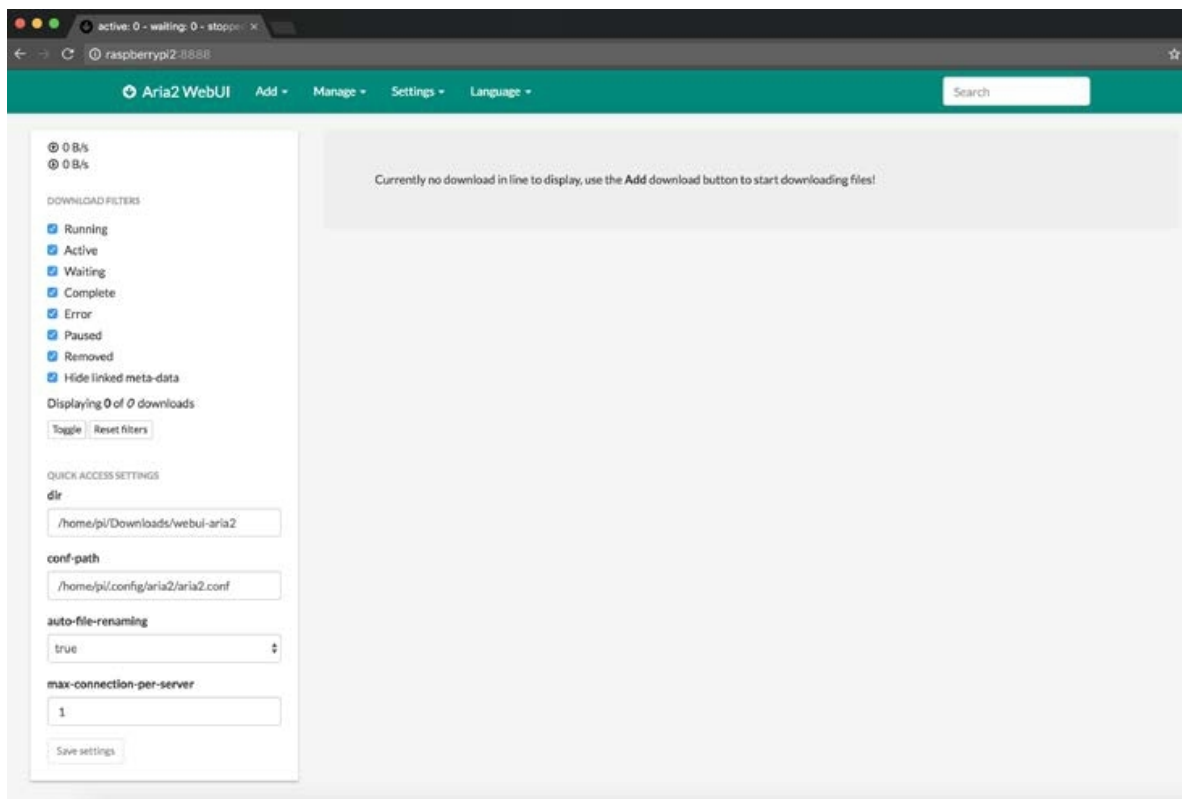


图37-1 WebUI界面

开始下载一个文件，只需要点击顶部绿色菜单栏中的Add菜单，可以选择从URL链接下载（By URL）或者通过种子文件下载（By Torrent）。

设置下载地址。在WebUI左侧的配置窗口中有一些配置选项。

- dir：树莓派上的下载地址，默认是Aria2运行的目录。
- conf-path：默认Aria2的配置文件地址，默认在当前账号 *Home* 目录的 *.config* 文件夹下。

树莓派自身的文件系统大小受限于SD卡的大小。如果想下载更大的文件，就需要给树莓派增加移动硬盘了。如果移动硬盘的文件格式是常见的ExFAT，则需要给树莓派安装ExFAT文件格式的支持。

```
$sudo apt-get update
$sudo apt-get install exfat-fuse exfat-utils
```

安装完这两个工具后，把移动硬盘拔掉再插上，它就可以被正常识别了。

在webui-aria2左侧有一个dir选项，把它换成移动硬盘的地址（默认是 `/media` 开头），就可以下载到移动硬盘上了。正在下载中的任务，如图37-2所示。



图37-2 正在下载中的任务

现在，我们有了一个容量大、有图形化支持、可以远程控制的离线下载工具了。

[1] Aria2的源代码在GitHub上，GitHub是目前全球最大的软件源代码托管仓库之一。Aria2软件的下载地址是<https://github.com/aria2/aria2/releases/tag/release-1.31.0>。

第38章 访客登记系统

在举办重大活动的时候，我们往往需要一个访客系统来记录每一位到访的客人。小巧而便于移动的树莓派，正适合这样的临时性场合。本章将介绍如何用树莓派制作一个小型的访客登记系统。这个系统具有以下功能。

- (1) 登记访客信息。
- (2) 打印访客名片^[1]。
- (3) 拍摄访客照片。

38.1 编写命令行小程序

用终端中的文本互动方式来实现访客登记系统。首先了解一下需要的基本编程工具Python。

1. Python 命令行程序语法

我们需要一个小程序用于登记来访客人的信息。在这一节中，我们用最基本的Python语法来编写这个程序。

这里使用Python 2.7版本，基本语法如表38-1所示。

表38-1 基本语法

功 能	写 法	说 明
获取用户输入	<code>a = raw_input("message")</code>	显示提示信息“message”，将用户输入保存在变量a中

续表

功 能	写 法	说 明
简单条件判断	<code>b = 1 if a == "yes" else 0</code>	如果变量a的值是字符串“yes”，则将变量b设置成1，否则将变量b设置成0
文本输出	<code>print "输出的文本"</code>	在命令行中打印print之后的字符串

例如，用下面的Python代码可以获取用户输入的一串文字。

```
your_input = raw_input("请输入一段文字：")
print your_input
```

这段代码包含了最基本的输入和输出。第一行是获取用户输入的文字，放进一个叫your_input的变量中，第二行输出用户输入的内容。

用上面这几个语法，我们就可以实现一个最基本的命令行访客登记系统。

2.实现访客登记系统

首先在树莓派的 *Home* 目录下创建一个名为 *visit.py* 的文件，在文件中输入以下内容。

```
#!/usr/bin/envPython
# -*- coding: utf-8 -*-

name = raw_input("请输入您的姓名：")
gender = raw_input("请输入您的性别（M/F）：")
gender = "先生" if gender == "M" else "女士"

print "您好，%s%s！" % (name, gender)
```

文件第一行，即#!/usr/bin/envPython，说明这个文件是python脚本。这一行如果是将文件直接用Python解释器执行（即下文中用Pythonscript.py命令来执行文件），是可以省略的。

文件的第二行，即#-*- coding: utf-8-*-，说明这个源代码是由UTF-8编码的。这样我们就可以在源代码中直接输入中文字符串了。

接下来的两行是获取用户的两个文本输入，分别是用户的姓名和性别。再下一行会根据用户的性别，即M或者F来设置“先生”或者“女士”的称呼。

最后一行是使用print在命令行中输出欢迎信息。这里输出的欢迎信息使用了文本格式化命令。

```
"您好，%s%s！" % (name, gender)
```

这个命令会用name和gender两个变量的内容依次替换前面字符串的“%s”。我们可以在 **Home** 目录运行这个文件。

```
$Pythonscript.py
```

运行效果如下，其中下画线部分是用户输入的内容。

```
请输入您的姓名: 周星星
请输入您的性别 (M/F): M
您好, 周星星先生!
```

在这个程序中，我们实现了基本的输入、输出功能。

其中，输入功能使用的是raw_input函数，这个函数可以接受一个参数，这里给的参数是“请输入您的姓名：”这个参数会被当作输入命令的提示显示出来。输出功能使用的是print功能，print语句后面可以跟随若干个字符串。这里输出的是“您好，名字+性别！”。

我们在这个小程序中还实现了一个判断功能。

```
gender = "先生" if gender == "M" else "女士"
```

这个语句的意思是，如果输入的gender是M，则把gender重新设置成“先生”，否则设置成“女士”。

38.2 尝试Tkinter

虽然文本互动的实现方式很简便，但是图形化的互动对于用户更友好。现在用Python下的图形化库Tkinter来实现图形化的访客系统。

Tkinter是编写PythonGUI（图形界面）程序最常用的工具，它提供了基于Tk的图形化界面开发接口。树莓派上的Linux操作系统也自带了Tkinter。编写Tkinter程序非常简单，下面是一段最简单程序的例子。

```
#!/usr/bin/envPython
# -*- coding: utf-8 -*-

from Tkinter import *

window = Tk()
window.title("Hello World!")
window.geometry('500x500')

label_name = Label(window, text="你好世界! ")
label_name.pack()

window.mainloop()
```

在上面的代码中，创建了一个程序窗口window。把这个窗口的title设置成了“Hello World!”，窗口大小设置成长500像素、宽500像素。

对Tkinter有了初步了解，我们就可以实现更复杂的图形化界面。下面这段代码可以在界面中添加一个输入框。

```
label_name = Label(window, text="姓名: ")
label_name.pack()
value_name = StringVar()
textbox_name = Entry(window, textvariable=value_name)
textbox_name.pack(fill=X)
```

输入框效果如图38-1所示。

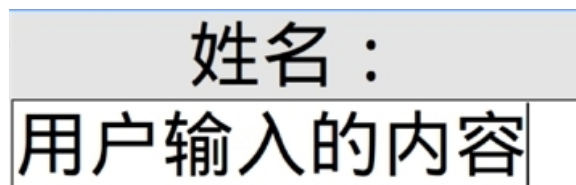


图38-1 输入框

这段代码给界面添加了两个控件，第一个是文字标签Label，第二个是输入框Entry。

创建文字标签的命令是Label(window,text="姓名: ")。这里可以看到，把文字标签的内容设置成“姓名：”之后，调用pack命令，即

label_name.pack(), 把文字标签放在窗口中。

创建输入框时，首先创建了一个文本值 StringVar，方法是 value_name=StringVar()，这样用户输入的内容就会被保存在这个文本值中。然后用 Entry(window,textvariable=value_name) 创建输入框。最后调用 pack 命令将输入框放在界面中。不同的是这次增加了一个参数 fill=X，这会让这个输入框横向占满窗口，让输入框的输入空间最大化。

为了实现性别输入，还需要制作选择框，代码如下所示。

```
label_gender = Label(window, text="性别: ")
label_gender.pack()
value_gender = StringVar()
Radiobutton(window, text="男", variable=value_gender, value="先生").pack()
Radiobutton(window, text="女", variable=value_gender, value="女士").pack()
```

这段代码中创建的文字标签 Label 与上面输入框并无本质差别。为了实现选择功能，我们在界面中做了两个可选按钮 RadioButton，分别显示成“男”和“女”，对应的选择值是“先生”和“女士”。

常见的用户界面还会有一些按钮。下面的代码实现了按下一个按钮然后弹出一个对话框的功能。

```
import tkinter as tk

def on_click():
    message = "您好! "
    tkMessageBox.showinfo(title='对话框标题', message=message)

button = Button(window, text="登记", command=on_click)
button.config(height=50, width=200)
button.pack()
```

这段代码引入了一个新的对象 tkMessageBox，它用来显示弹出窗口。接下来的代码为两部分，第一部分是一个 **回调函数**（Callback Function），名字是 on_click。这个函数的功能是弹出一个对话框，对话框上写着“您好”，如图38-2所示。

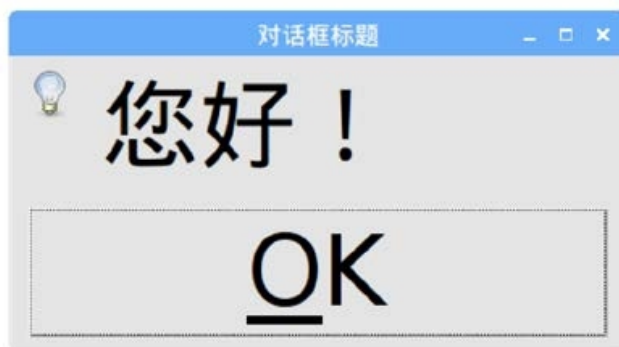


图38-2 弹出对话框

第二部分是制作一个按钮，我们把这个按钮的文字设置成了“登记”，命令设置成了上面定义的回调函数on_click，再设置了它的高度和宽度，最后用pack命令放置在主窗口上。我们可以使用下面这段代码来自定义字体和字号。

```
import tkFont

font = tkFont.nametofont("TkDefaultFont")
font.configure(size=48)
window.option_add("*Font", font)
```

这段代码把字体设置成为了Tkinter的默认字体TkDefaultFont，把字号设置成48号。

38.3 制作访客登记系统

把刚才的功能结合起来就可以制作一个完整的访客登记系统。完整的程序代码如下所示。


```

#!/usr/bin/envPython
# -*- coding: utf-8 -*-

from Tkinter import *
import tkMessageBox
import tkFont

window = Tk()
window.title("访客登记系统")
window.geometry('500x500')

font = tkFont.nametofont("TkDefaultFont")
font.configure(size=48)
window.option_add("*Font", font)

label_name = Label(window, text="姓名: ")
label_name.pack()
value_name = StringVar()
textbox_name = Entry(window, textvariable=value_name)
textbox_name.pack(fill=X)

label_gender = Label(window, text="性别: ")
label_gender.pack()
value_gender = StringVar()
Radiobutton(window, text="男", variable=value_gender, value="先生").pack()
Radiobutton(window, text="女", variable=value_gender, value="女士").pack()

def on_click():
    message = "您好, %s%s! " % (value_name.get().encode('utf-8'),
value_gender.get().encode('utf-8'))
    tkMessageBox.showinfo(title='感谢使用访客登记系统!', message=message)

button = Button(window, text="登记", command=on_click)
button.config(height=50, width=200)
button.pack()

window.mainloop()

```

运行效果如图38-3所示。



图38-3 访客登记系统主界面

38.4 访客名片和访客拍照

加入打印访客名片的功能，使用ESC/POS指令来打印名片，打印名片可以贴在访客身上作为名牌。ESC/POS是Epson Standard Code/Point of Sale的缩写，它的本意是爱普生标准代码/收银系统。因为这个协议十分简单清晰，所以它被绝大部分市面上出售的不同品牌小票打印机或者类似的小型打印机支持。

市面上的小票打印机很多都提供USB接口。USB是现在最常用的数据接口格式之一，刚好树莓派也自带了USB接口。这里强烈推荐读者使用带USB接口的POS打印机。

将打印机通过USB连接到树莓派后，可以通过路径/dev/usb/lpX访问它。路径最后的X是从0开始的编号，代表当前连接的USB设备。具体这个数字多少，最简单的方法是从0开始尝试，如果可以打印就表示成功了。

用Python向打印机接口输出文字的方法如下所示。

```
printer = '/dev/usb/lp0'

def printline(line):
    with open(printer, 'w') as f:
        f.write(line.decode('utf8').encode('gb2312') + '\n')
        f.flush()
```

这段代码先假设打印机的路径是/dev/usb/lp0，然后定义了一个函数printline()。将这段代码放在Python代码中，调用printline函数就可以让打印机打印文字了。

注意，因为常见的中文小票打印机使用的是GB2312编码，而Python程序是UTF-8编码的。如果打印的内容是中文，则需要先用UTF-8格式解码，再用GB2312格式编码，所以要用line.decode('utf8').encode('gb2312')函数。

有了这个函数就可以编辑上一部分的代码，将用户的名字和称呼打印出来。

```
printline("名片: %s %s" % (name, gender))
```

访客系统可以结合第13章介绍的树莓派拍摄来采集访客照片。这里可以用Python来控制摄像头捕捉照片。

```
import subprocess
subprocess.call(["raspistill", "-o", "%s.jpg" % name])
```

这段代码调用的是Python的subprocess库。第二行实际上相当于在命令行中执行：

```
$raspistill -o 访客名字.jpg
```

这个命令的效果就是按照访客的名字把他们的照片保存下来。到了这里，一个功能多样的访客系统就诞生了。

[1] 打印访客名片的功能是通过一台POS打印机实现的。如果没有POS打印机，那么可以去电商平台搜关键字“小票机”购买，最便宜的打印机不到百元。当然，你也可以跳过打印名片部分。

第39章 节能照明系统

树莓派不仅可以当作一台普通电脑使用，而且因为它有丰富的电路接口，可以跟很多硬件整合，做更多有创意的事情。本章介绍如何利用树莓派制作一个节能照明系统。节能照明系统需要一个光照传感器来检查太阳光线，以及一个继电器控制照明电路。当树莓派检测到周边光照弱时，就打开照明灯。

39.1 传感器

1. 模拟信号与数字信号

传感器（Sensor）是用来测量某一变量并将结果转化为电信号的设备，而电信号分为模拟信号和数字信号。模拟信号指电路中用电压表示的信号。假如有一个模拟信号的温度计，它的输出电压范围是0到5伏，可以测量0°C到100°C的温度，那么可以规定一个线性的电压和温度对应，如表39-1所示。

表39-1 电压和温度对应表

电压（伏特）	表示温度（摄氏度）
0	0
2.5	50
5	100

不过，通常模拟信号的传感器和测量值不是线性对应关系。我们需要进行一个数学计算来求实际测量值，这个数学计算可以用数学公式表示。例如，这个线性关系的数学公式如下所示。

$$\text{温度} = \text{电压} \times 20 \text{（摄氏度/伏特）}$$

数字信号和模拟信号不同，它是现代电子计算机使用的信号表达模式。与模拟信号不同，数字信号传递的数值是绝对精确的。数字信

号通常使用二进制来表示数。假如有一个可以测试0°C到100°C的温度计，现在测量到的温度是36°C，36转换成二进制数是100100。我们只需要传播100100这个数字即可。

定义一个数字信号，首先要将电压范围划分成高低两部分，这样就可以没有歧义地传递一个信息，即电压的高或者低。一个5V的数字信号通常会把3.5V~5V定位为高电平，0V~0.25V定位为低电平，高电平对应1，低电平对应0，而0.25V~3.5V属于中间范围，在正常情况下不会出现，避免误差造成错误结果。定义好1和0后，人们还会规定这个数字信号的频率。信号发出者会按照这个频率依次传递0或者1的数据，而数据接收者会根据这个频率来读取0或者1。将读取到的数据连接起来，就可以得到整个二进制数据了。

2.传感器的接口

因为树莓派是一个纯数字的设备，所以它可以直接读取数字传感器的读数。在这个例子中，需要读取当前是否有太阳光照，最简单的方法就是使用一个数字光传感器。我们将使用一个数字光敏传感器，这个传感器有3个接口，分别是VCC、GND和DO（Digital Output）。通常接口附近一般都会有白色油漆印的小字。

- VCC，供电接口。树莓派提供3.3V和5V的电压输出，传感器使用的是5V电压。
- GND，地线。
- DO，数字输出接口。

这个光敏传感器只有两种输出状态：1代表有光，0代表无光。将该传感器连接到树莓派的GPIO端口。光敏传感器上的VCC、GND和DO三个接口分别接在树莓派上的5伏电源、GND和任意一个GPIO口，例如GPIO2。GPIO接口可参考第11章。

39.2 读取传感器数据

将传感器与树莓派连接好就可以在树莓派上读取传感器的结果了。树莓派的GPIO模块名叫RPi.GPIO。如果这个模块没有被安装，需

要用pip来安装这个模块，在命令行中执行下面这个命令。

```
$pip install RPi.GPIO
```

新建一个Python脚本文件，如果要在 *Home* 目录下新建一个名叫 *light.py* 的Python脚本文件，则可以使用下面的命令。

```
$cd ~  
$touch light.py
```

Python脚本的第一行可以声明这是一个Python脚本。

```
#!/usr/bin/env python
```

设置端口为输入或输出。如果把号码为2的端口设置为输入端口，则需要做：

```
GPIO.setup(2,GPIO.IN)
```

如果把号码为2的端口设置为输出端口，则是：

```
GPIO.setup(2,GPIO.OUT)
```

可使用下面的语句读取2号端口的数值：

```
GPIO.input(INPUT_PIN)
```

结合上面的几点，就可以制作一个小程序，每秒打印当前光敏电阻的读数。

```
import RPi.GPIOasGPIO
```

下面设置两个常数。

```
SLEEP_TIME = 1 # 检查读数的频率, 1 代表 1 秒
INPUT_PIN = 2 # 读数的接口, 2 代表 BCM 2 接口

GPIO.setup(INPUT_PIN,GPIO.IN)
```

定义一个函数来获取光敏电阻读数。

```
def get_light_reading():
    return GPIO.input(INPUT_PIN)
```

主程序如下：

```
def main():
    while True:
        light_reading = get_light_reading()
        print light_reading
        time.sleep(SLEEP_TIME)
```

一个小技巧，当按快捷键 Ctrl+C 时，程序会退出并执行 GPIO.cleanup()。

```
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        GPIO.cleanup()
```

39.3 控制照明电路

根据传感器感应的光照情况来控制照明电路。

1. 继电器

在第11章中已经介绍过如何用树莓派点亮一盏LED小灯，小灯直接连接的是树莓派提供的5V电压。但是常见的照明电路连接的是220V

的电压，这就需要使用树莓派的5V电路来控制一个开关，通过这个开关可以控制照明灯的开关。这种可以被信号电路控制的开关被称为继电器。

继电器上有3个低压控制接口，分别是VCC、GND和DI。其中VCC、GND与前面的传感器一样，DI是数字输入接口。继电器上还有两个高压接口，它根据数字输入接口的输入联通或者断开。

将继电器与树莓派连接好，其中DI接口连接GPIO的BCM3接口。将照明电灯连接电路（单相交流电）的火线切断，分别接在继电器的两个高压接口上。注意，照明电路的电压远高于人体安全电压，操作的时候必须断电，一定要注意安全。连接好后就可以使用树莓派来控制继电器了。

2.Python控制GPIO继电器

用Python控制GPIO继电器和读取传感器读数非常类似。区别在于我们把接口的模式改成OUT（输出）。再用一个if条件句，设置当没有光照时给继电器发出1（接通）信号，否则发出0（断开）信号。具体代码如下所示。

```
import RPi.GPIO as GPIO
import time

SLEEP_TIME = 1
INPUT_PIN = 2
OUTPUT_PIN = 3

GPIO.setmode(GPIO.BCM)
GPIO.setup(INPUT_PIN, GPIO.IN)
GPIO.setup(OUTPUT_PIN, GPIO.OUT, initial=GPIO.LOW)

def get_light_reading():
    return GPIO.input(INPUT_PIN)

def set_light_status(is_on):
```



```

    if is_on:
        GPIO.output(OUTPUT_PIN,GPIO.HIGH)
    else:
        GPIO.output(OUTPUT_PIN,GPIO.LOW)

def shall_turn_light_on(light_reading):
    return light_reading < 30

def main():
    while True:
        light_reading = get_light_reading()
        light_status = shall_turn_light_on(light_reading)
        set_light_status(light_status)
        time.sleep(SLEEP_TIME)

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        GPIO.cleanup()

```

这样，完整的节能光照控制系统的程序和电路就完成了。这个系统会在传感器检测不到光照时点亮照明灯，否则关闭照明灯。

第40章 树莓派挖矿

比特币和区块链是这几年热门的话题。比特币是一种网络货币，它不属于任何一个国家，加密并去中心化。比特币的上涨和下跌也让几家欢喜几家愁。比特币可以通过一种名为“挖矿”的计算过程产生。负责挖矿的计算机称为矿机。本章将介绍树莓派在比特币和区块链方面的应用，把树莓派改造成一个低功耗的矿机，让它在家中静静地创造财富。

40.1 比特币钱包

在进行挖矿之前，需要一个比特币钱包来存放已有的比特币。比特币钱包可以分为两种。

一种是比特币交易平台提供的“钱包”。这种“钱包”事实上是用户在交易平台上创建的比特币账户。这些比特币实际存放在交易所的比特币账户上，它的安全性由交易平台担保。用户不能自己使用密钥支配钱包中的比特币，但可以通过交易所提供的网站或手机App操作。

使用简便、不需要计算机知识是这种“钱包”最大的好处。用户不需要理解比特币的技术细节，只需要像购买基金、股票等证券产品一样使用网络系统即可。但是这种“钱包”的最大问题是，事实上它不具备比特币的分布式和去中心化特点，用户的资金安全取决于交易所的信用。

图40-1是知名比特币交易平台coinbase的主界面，它向用户提供了在线的比特币钱包服务。

另一种比特币钱包需要用户自己保存比特币的密钥，或者是解锁密钥的**密码**（Passphrase）。这种比特币钱包非常多，它们可以运行在不同的操作系统、浏览器上，甚至还有硬件的比特币钱包，安全性更强。

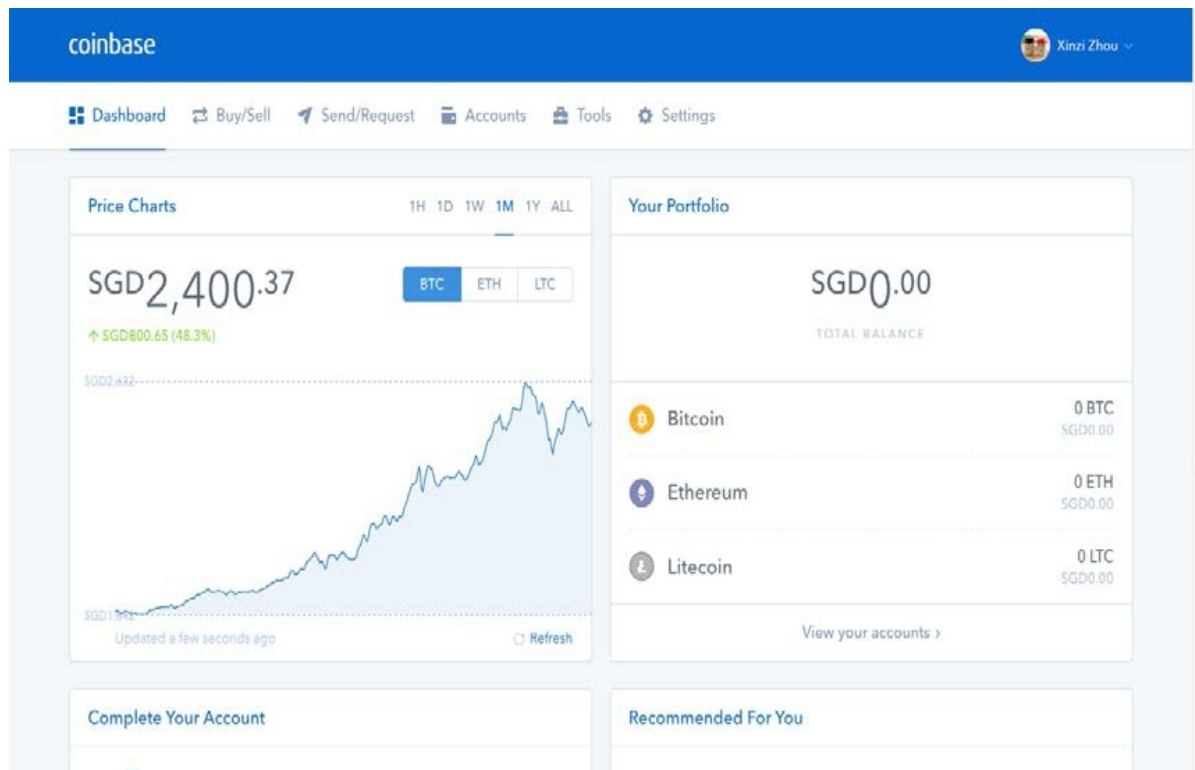


图40-1 比特币交易平台coinbase

比特币官方给出了一个比特币钱包软件的列表^[1]。这里介绍一个名叫Electrum的钱包软件，它可以被安装在树莓派中，安装方法十分简单。首先，在树莓派上安装Python相关的一些包。

```
$sudo apt-get install python-qt4 python-pip
```

然后，用下面一行指令使用pip2安装Electrum。

```
$sudo pip2 install https://download.electrum.org/2.8.2/Electrum-2.8.2.tar.gz
```

安装完后，在命令行输入electrum命令，就可以启动Electrum程序了。第一次运行Electrum会看到安装引导界面。我们按照默认选项安装，直到生成一个12个单词的随机种子，如图40-2所示。这12个单词是打开和恢复你的比特币钱包最重要的东西，一定要好好保存。下一步，Electrum会要求用户输入这12个单词，以确保用户把它们妥善地保存了下来。

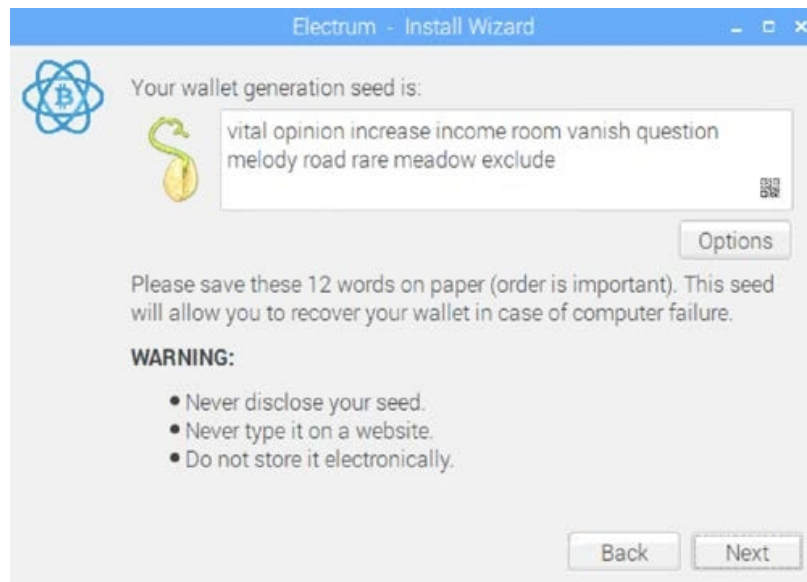


图40-2 12个单词的随机种子

最后一步是设置钱包的密码，这个密码用来加密你的比特币钱包的密钥。测试时，我们可以把这个密码留空。至此，Electrum钱包创建完成。进入Electrum的主界面，如图40-3所示。

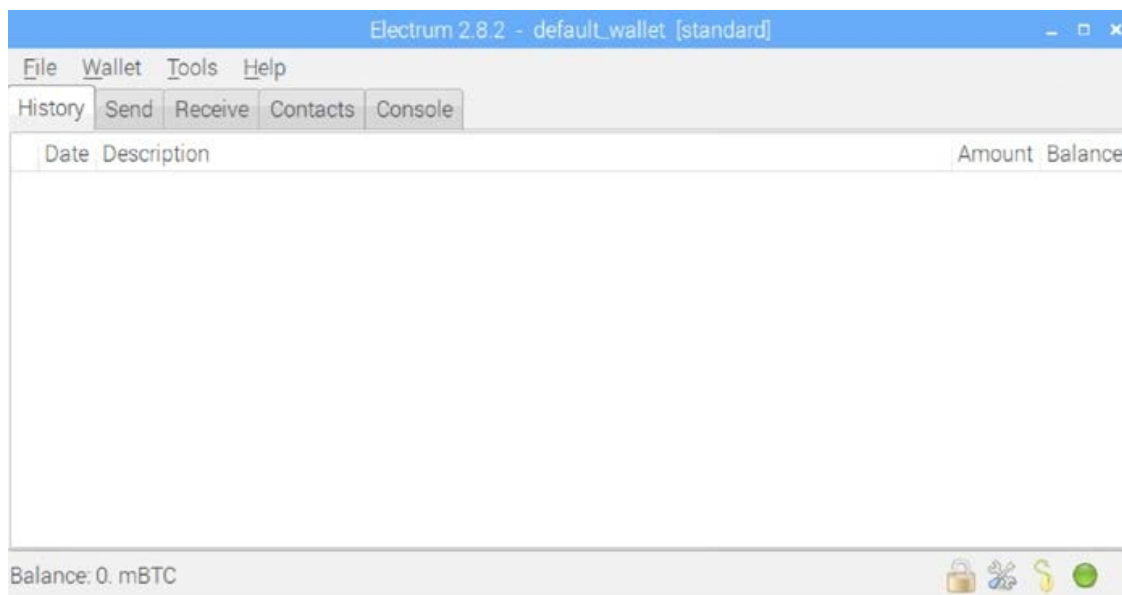


图40-3 Electrum钱包主界面

40.2 在树莓派上挖矿

比特币每10分钟产生一个合法区块，负责产生这个区块的矿机将获得比特币奖励。根据比特币的设计，所有的参与者会参与一场运算竞赛。在这10分钟内全世界只有一个矿机胜出，独享这个区块对应的比特币，即成功挖出比特币。

在比特币诞生的早期，单台计算机还很有希望挖出比特币。但目前来说，单台计算机想要挖掘出比特币已经十分困难了。于是就有很多挖矿的人联合起来建立矿池。一个矿池通常有成千上万台计算机在同时挖掘。一个矿池中只要有一台计算机成功挖到区块，所有挖矿者将会根据自己的贡献得到相应的提成。我们的树莓派就将加入这样的矿池中挖矿。树莓派需要使用一个名为BFGMiner的挖矿软件，它可以连接Slushpool矿池。

在安装BFGMiner之前，先到Slushpool^[2]上注册一个账号。注册完成后，登录Slushpool网站，进入Workers选项，找到默认创建的worker1。单击worker1->Edit Worker就可以看到图40-4的界面。

Edit Worker

Miner login dreamrunner.worker1
The password can be any arbitrary text

Worker Name

worker1

7/32

Labels

☒ Enable monitoring

The monitoring system will detect and report any hash rate issues related to this worker.

☒ Auto detect Alert Limit

Automatically estimates Hash Rate of your worker by its past activity.

☒ Use default minimum difficulty

User minimum difficulty from your user profile settings.

Alert Limit [Gh/s]

5

The system will report issues, when hash rate drops below this value.

Minimum Difficulty

128

Pool always assigns difficulty equal or greater than this value.

Delete

Revert

Cancel

Save

图40-4 修改工人页面

在修改工人（Edit Worker）页面的右上角可以找到挖矿工人的登录账号，图40-4上是dreamrunner.worker1，登录密码可以是任何字符串。登录密码之所以不受限制，是因为别人拿到你的账号，也只能帮你挖矿。因此，即使被盗号，也不是坏事。

打开树莓派的命令行用git下载BFGMinor的源代码。

```
$git clone https://github.com/luke-jr/bfgminer.git
```

BFMiner软件下载到当前目录的一个文件夹下，下面的命令将会安装这个软件。

```
$cd bfgminer  
$./autogen.sh  
$./configure  
$make
```

通过下面的命令来运行BFGMiner，把其中的username、worker和password替换成自己在Slushpool上注册的信息即可^[3]。

```
$/bfgminer -o stratum.bitcoin.cz:3333 -O username.worker:password -S all
```

还可以把树莓派挖矿的程序设置为开机启动。一旦开机，树莓派将会自动挖矿，打开树莓派的命令行，使用nano来编辑 */etc/rc.local* 文件。

```
$sudo nano /etc/rc.local
```

在这个文件的末尾添加刚才启动BFGMiner的脚本就可以让它开机启动。

注意，启动脚本中的 *./bfgminer* 需要替换成这个二进制文件的绝对路径。

40.3 区块链存储服务

比特币背后的技术就是区块链。它的本质是连续增长的记录链条，每一个记录被称为“区块”。区块链技术可以安全地储存、添加和修改数

据，其安全性有密码学理论支持。比特币只是区块链的一种。下面介绍基于区块链的分布式储存服务，即Storj。

Storj是一个分布式的文件存储服务提供商。与Dropbox、百度网盘类似，它向用户提供文件存储服务，但是Storj并不把用户的文件存在自己的服务器上，而是把文件存储任务交给互联网上的匿名参与者。文件的所有者不用担心文件的安全，因为文件是被区块链技术加密后才发给文件存储者的。作为回报，给Storj用户提供文件存储服务的人将会收获一种Storj发行的基于以太坊区块链的电子货币，这种方式被Storj称为Storj Share。

比特币矿池的存在降低了比特币挖矿的难度，但是树莓派的计算能力确实有限。想要让树莓派通过挖矿获取比特币十分困难。虽然树莓派功耗极低、计算能力有限，但是可以连接数千太字节移动硬盘，因此特别适合用来提供Storj的存储服务。树莓派通过Storj Share向其他人提供文件存储服务，从而获取利润。

我们需要一个以太坊的钱包来存储Storj的回报，和比特币一样，获得以太坊钱包的方法有很多，这里介绍一个My Ether Wallet的在线服务。首先，打开My Ether Wallet在线服务的网页^[4]，如图40-5所示。



图40-5 MyEtherWallet的网页

在首页输入一个自己一定不会忘记的密码。这个密码将会在今后使用钱包的时候用到。因为My Ether Wallet本身并不储存密码，所以一旦忘记了这个密码，钱包中的以太坊将无法使用。

创建好钱包后，会得到一个钱包地址和一个密钥，如图40-6所示，这个钱包地址就可以被用来接收STORJ Token。



图40-6 My Ether Wallet的钱包页面

安装Storj Share程序。Storj官方推荐使用nvm来安装node.js的运行环境，安装nvm的方法如下。

第一步，打开命令行执行下面这行命令。

```
$wget -q0-https://raw.githubusercontent.com/creationix/nvm/v0.33.3/install.sh |  
bash
```

关闭当前命令行窗口，并打开新的命令行窗口。这样做是为了在命令行中激活nvm命令。下面就可以安装LTS（长期支持）版本的node.js了。

```
$nvm install -lts
```

此外，还需要安装一些系统工具。

```
$sudoapt-getinstall -y git python build-essential
```

这样就可以使用node.js的包管理软件npm来安装Storj Share的程序storjshare-daemon。

```
$npm install --global storjshare-daemon
```

安装好就可以开始运行storjshare daemon了。


```
$storjshare daemon
```

第二步，生成Storj Share的配置文件。创建配置文件需要使用下面的命令，把其中的YOUR_STORJ_TOKEN_WALLET_ADRESS替换成刚才创建的My Ether Wallet地址，把~/storj.io/修改为你希望用来给Storj存放文件的路径：

```
$storjshare create --storj=YOUR_STORJ_TOKEN_WALLET_ADRESS --storage=~/storj.io/
```

执行这个命令后，程序会输出下面文字（下面JSON配置文件的文件名是随机的）：

```
* configuration written to
/home/pi/.config/storjshare/configs/5eefab39cac083daabd9e15d49c2ce0eeba901c4.json
* opening in your favorite editor to tweak before running
...
* use new config: storjshare start --config
/home/pi/.config/storjshare/configs/5eefab39cac083daabd9e15d49c2ce0eeba901c4.json
```

根据上面的最后一行来运行Storj Share，即在命令行输入：

```
$start --config
/home/pi/.config/storjshare/configs/5eefab39cac083daabd9e15d49c2ce0eeba901c4.json
```

这样，Storj Share就在树莓派上顺利运行了。利用树莓派，我们就可以参与Storj Share的区块链经济中去了。

[1] 网址<https://bitcoin.org/en/choose-your-wallet>。

[2] Slushpool的网址<https://slushpool.com/>。

[3] 笔者的用户名就是dreamrunner.worker1，密码任意。

[4] 网页地址<https://www.myetherwallet.com/>。

第41章 高性能计算

信息时代需要高性能计算能力。说到高性能计算，我们往往指的是分布式计算，也就是让多个处理器或计算机合作，共同提高计算机的计算效率。小小的树莓派也可以组成一个分布式计算的集群。虽然每个树莓派的运算性能有限，但是将多台树莓派组合起来、配合高效的算法就可以实现性能上的巨大提升。本章将用两台树莓派作为例子，将它们组成一个分布式计算集群，并验证集群比单台树莓派可以更快地计算出同一个问题的答案。

41.1 Spark

让多台计算机合作是一个复杂的过程。幸运的是，我们可以直接使用Spark这样的工具。Spark是UC Berkeley大学AMP实验室开发的通用分布式计算框架。Spark与Hadoop类似，Hadoop是另一个并行计算框架，由Apache基金会开发维护。Spark和Hadoop都使用了MapReduce的编程模型。

MapReduce常用在分布式数据处理中，这个编程模型把数据处理分成Map和Reduce两步。

- Map步骤：将输入数据转换成为大量的**键值对**（Key-Value Pair）。其中的**键**（Key）会被当作归类键值对的根据。也就是说，相同键的键值对会被放在一起。执行Map步骤的程序被称为Mapper。

- Reduce步骤：把相同键的键值对的值进行**聚合**（Aggregation），输出一个相对简单的结果。执行Reduce步骤的程序被称为Reducer。

例如，统计一本书中每个单词出现的频率。一种编程模型是直接的编程方式，它用循环的方式来遍历每个词，再统计词频。MapReduce则是不同的思路。在Map步骤中，Mapper就会将这本书的每个单词提取出来，生成（单词，1）这样的键值对，而在Reduce步骤中，Reducer会将同样键的值相加，相加的结果就是所谓的键，也就是某个单词的出现次数。MapReduce分割了任务，从而允许让不同的电脑充当Mapper和

Reducer。比如在图41-1的过程中就可以有三台电脑分别进行Mapping任务，每台电脑负责一句话的词频统计。这样，就可以更快地统计词频。

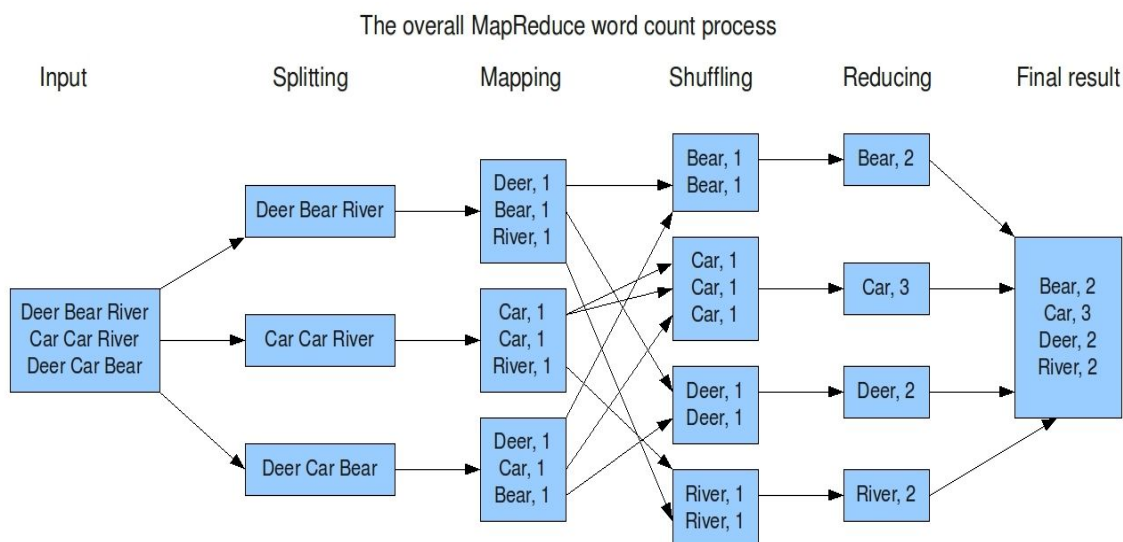


图41-1 MapReduce任务

41.2 树莓派与Spark

Spark的运行环境需要Java虚拟机和Python解释器。树莓派自带的Raspbian OS上自带了两者，不需要额外安装。

在树莓派上使用Spark，只需要从官网下载已经编译好的版本即可。首先，打开Spark官网的下载页面^[1]，找到最新的下载链接。然后，通过浏览器或者wget工具来下载。如果使用wget下载，命令如下：

```
$cd ~  
$wget https://d3kbcqa49mib13.cloudfront.net/spark-2.1.1-bin-hadoop2.7.tgz
```

这会把一个tgz的压缩包下载到当前目录，用tar命令来解压。

```
$tar-zxvf spark-2.1.1-bin-hadoop2.7.tgz
```

这样就下载好Spark程序了，进入Spark程序的路径。

```
$cd spark-2.1.1-bin-hadoop2.7
```

为了方便，我们把Spark路径修改成~/spark，也就是/home/pi/spark：

```
$mv ~/spark-2.1.1-bin-hadoop2.7 ~/spark
```

41.3 单机版 π 计算

安装完成后，我们可以检验树莓派的运算能力。我们选择的计算问题是 π 的计算。 π 是圆周率，代表圆的周长和直径的比例。它是一个无限不循环的无理数，通常用3.14来近似表示。使用Spark的分布式计算来提高计算机估算 π 的速度。

Spark自带了计算圆周率 π 的示例程序。该程序的计算原理非常简单。这个程序会随机选择在一个面积为 2×2 的正方形中的一个点，并不断重复这个过程。最后，程序统计点落在半径为1的正方形内切圆的概率。正方形和它的内切圆的几何关系，如图41-2所示。

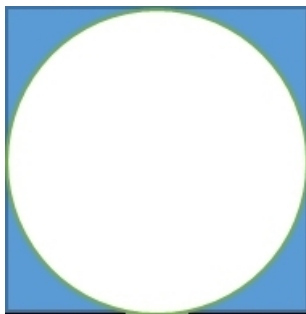


图41-2 正方形和它的内切圆

点落在内切圆的理论概率是：

$$\frac{S_{\text{圆}}}{S_{\text{正方形}}} = \frac{\pi \times 1^2}{2^2} = \frac{\pi}{4}$$

如果模拟得到的点落在内切圆的比例是 x ，那么就可以得到等式：

$$\frac{\pi}{4} = x$$

这样就可以反推出 π 的值：

$$\pi = 4x$$

随着随机取点次数的增多，估算的 x 值也就越接近理论值。下面进行20次模拟来估算 π ，命令如下：

```
$bin/run-exampleJavaSparkPi 20
```

开始运行后，命令行上很多运行**日志**（log）被打印出来。它们的格式与下面这行类似：

```
17/05/14 07:20:02 INFO DAGScheduler: Job 0 finished: reduce
atJavaSparkPi.java:52, took 19.255536 s
```

日志的格式是<日期时间> <日志级别> <日志类别> <日志内容>。这行日志说的是第0号任务完成，执行任务花费了大约19秒。日志级别是INFO（信息）。

这个程序最后的输出也会被打印出来：

```
Pi is roughly 3.140356
```

值得注意的是，上面的Spark只运行在一台树莓派上。下面将增加树莓派来见证分布式运算的威力。

41.4 树莓派集群

Spark集群是指将多台计算机连接起来，一起运行同一个Spark程序，从而得到更快的速度，处理更多的数据。本节介绍Spark集群搭建的方法。这个集群有两台树莓派，分别被称为树莓派1和树莓派2。如果没有特殊说明，我们的指令默认在树莓派1中执行。

1.准备

设置树莓派的无密码访问。在Spark集群中，经常需要在一台树莓派上控制别的树莓派，如果每次都需要输入密码，那么这个步骤会显得尤为烦琐。

首先，在树莓派1中生成一份公钥和密钥，用来访问别的电脑。

```
$ssh-keygen -t rsa -C "你的邮箱地址"
```

这个脚本会在你的树莓派中生成两个文件。

- `~/.ssh/id_rsa`
- `~/.ssh/id_rsa.pub`

第一个文件是你的密钥，第二个文件是公钥。

然后，设置树莓派2，允许树莓派1使用这份公钥和密钥访问。把树莓派1的公钥放在树莓派2的 `~/.ssh/authorized_keys` 文件中。操作方法是，在树莓派1中将RSA公钥输出。

```
$cat ~/.ssh/id_rsa.pub
```

再通过ssh进入树莓派2，编辑文件 `~/.ssh/authorized_keys` 。

```
$nano ~/.ssh/authorized_keys
```

如果这个文件是空的，则将树莓派1的公钥粘贴在里面即可。如果这个文件本来有内容，则新建一行，将公钥贴在上面。

由此，树莓派1到树莓派2的无密码访问设置好了，可以用ssh命令测试是否需要输入密码。反向设置一下从树莓派2到树莓派1的无密码访问，方法和之前的完全一样。

设置树莓派的主机名。为了方便管理，给每一个树莓派都设置一个不同的主机名，即raspberrry和raspberrry2，这样不需要用IP地址也可以访问这两个树莓派了，更加方便管理。例如可以用下面的命令来访问树莓派1。

```
$ssh pi@raspberrry
```

通过更改 `/etc/hostname` 文件中的内容来改变主机名，其他更改主机名的方法可以参考第8章。

2.搭建集群

选择一台树莓派作为主控节点，用它控制所有Spark的**工人**（Worker）。在节点众多时，可能不会把主控节点本身设置为Worker。

但在这个例子中只有两个树莓派，因此两个树莓派都会被设置成 Worker。

编辑Spark目录下的 *conf/slaves* 配置文件。

```
$nano conf/slaves
```

将raspberry和raspberry2分别写入这个文件中，Spark就会自动使用这两个机器作为计算的工作节点。

随后，设置Spark的运行环境。在Spark的 *conf* 目录下，有一个叫 *spark-env.sh* 的文件，这个文件是Spark运行环境的设置。首先，把默认的配置文件的 *spark-env.sh.template* 复制一份保存在 *spark-env.sh* 中。

```
$cd conf
$cp spark-env.sh.template spark-env.sh
```

然后，修改 *spark-env.sh* 文件，将其改成我们需要的配置。其中最重要的两项设置如下所示。

```
SPARK_MASTER_IP=192.168.1.100
SPARK_WORKER_MEMORY=512m
```

SPARK_MASTER_IP是指**主控节点**（Master Node）的IP地址，这里我们填上树莓派1的IP地址。SPARK_WORKER_MEMORY是指Spark的工作节点中运行时使用的最大内存。树莓派3B版的内存是1GB，我们可以将SPARK_WORKER_MEMORY的值设置成512MB。

在树莓派1中完成设置后，还需要复制配置文件到树莓派2上。下面的语句会将配置文件复制到树莓派2上。

```
$scp conf/* pi@raspberry2:spark/conf/
```

3.在集群上运行程序

在树莓派1的spark目录中运行 *./sbin/start-all.sh*。这个脚本会启动控制节点和所有工作节点，启动完成后用浏览器打开<http://raspberrypi:8080/>将会看到控制页面。在正常情况下，页面会显示树莓派集群有两个 Worker，而SparkMaster在集群上成功运行。随后，我们可以用分布式计

算圆周率了。执行SparkPi程序的方法比之前稍微复杂了一点，要使用spark-submit脚本。

```

$./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://192.168.1.106:7077 \
  --executor-memory 512MB \
  --total-executor-cores 2 \
  /home/pi/spark/examples/jars/spark-examples_2.11-2.1.1.jar \
  20

```

这个命令设置了几个参数，如表42-1所示。

表42-1 参数

参 数	说 明
--class	Spark 主程序的 class
--master	主控节点的地址
--executor-memory	每个工作节点使用的内存
--total-executor-cores	总共使用的 CPU 核心数量

由于树莓派的内存大小有限，我们在搭建的拥有两个节点的Spark集群上只设置了两个**执行器**（Executor），每个执行器配备512MB的内存。注意，Spark规定每个执行器最少要有450MB的内存。

在这个配置中，两个树莓派节点将会各生成一个执行器共同执行这20个**任务**（Task）。理论上说，它的执行速度会比之前的单机版本快一倍，但是由于一些网络开销和无法并行的计算步骤，实际速度的提升一般都会低于理论速度。

[1] 官网的下载地址是<http://spark.apache.org/downloads.html>。

第42章 蓝牙即时通信

现代生活离不开即时通信技术。我们每天都在使用基于互联网的即时通信技术。不过，跳出互联网的限制，让树莓派以蓝牙通信的方式来实现即时通信。这个系统将可以实现文字的传输，使用蓝牙协议进行数据传输，完全不依赖互联网。传输内容经过加密，无法被人窃听或者篡改。这个例子中我们至少需要用到两个树莓派。在搭建系统的过程中，需要用网络下载依赖的软件。

42.1 树莓派与蓝牙

我们使用的蓝牙通信协议叫作RFCOMM。RFCOMM是一套基于L2CAP协议的简易数据传输协议。因为RFCOMM可以用蓝牙模拟两个设备间的RS-232连接（一种电脑上常用的串行数据接口），所以也可以把RFCOMM协议称为串口仿真。

为了使用RFCOMM协议，我们需要使用一些开发工具和库。BlueZ是Linux平台上的官方蓝牙工具集。它向开发者提供了模块化的蓝牙相关工具，从底层到蓝牙协议的各个协议层。而Pybluez是BlueZ的Python拓展，它让我们可以在Python脚本中使用Bluez的功能。使用Pybluez可以很轻松地用Python编写蓝牙程序。

先在树莓派上安装Pybluez，由于蓝牙开发涉及硬件，而不同计算机的硬件可能会有一些区别，所以安装过程中可能会遇到各种不同的问题。笔者在开发时，遇到了树莓派与蓝牙pnat拓展不兼容的问题，按照下面的方法禁用了pnat插件后，Pybluez就可以在树莓派上完美运行了。

第一步，打开命令行，使用apt-get安装Python和蓝牙需要的库和工具：

```
$sudo apt-getinstall python-dev libbluetooth-dev bluetooth bluez
```

第二步，使用Python的包管理工具pip安装Python包Pybluez：

```
$sudo pip install pybluez
```

树莓派上的蓝牙程序有时候会因为pnat功能而无法正常运行，所以要在树莓派上禁用pnat插件。用nano工具编辑 `/etc/bluetooth/main.conf` 文件：

```
$sudo nano /etc/bluetooth/main.conf
```

在这个文件里加入一行：

```
disableplugins = pnat
```

到此为止，树莓派上的蓝牙开发环境就配置好了。我们可以用下面简单蓝牙服务器程序的例子来测试安装是否成功。

42.2 蓝牙服务端

蓝牙协议分为两个角色，即服务端和客户端。服务端程序将会发出广播，等待客户端连接。客户端会搜寻可用的服务端，找到服务端后与服务端进行配对。客户端与服务端配对成功后，两者之间就可以开始传输信息。蓝牙服务端的程序需要具有下面的功能。

- 监听一个蓝牙通信通道。
- 使用Pybluez的advertise_service方法来让客户端可以找到我们的程序。
- 接受一个客户端的连接。
- 打印客户端发送的内容。
- 关闭和客户端的链接及蓝牙程序。

在蓝牙通信之前，首先让服务端产生UUID。Pybluez的advertise_service方法需要用到一个唯一的地址UUID。这是让客户端识别服务端的识别代码，也就是说，这是蓝牙服务的唯一名字，因此要生成一个唯一的UUID。

用Python生成UUID的方法很简单，打开Python交互命令行：

```
$python
```

引入uuid包：

```
>>> import uuid
```

运行uuid.uuid1()命令：

```
>>> uuid.uuid1()  
UUID('63078d70-feb9-11e7-9812-dca90488bd22')
```

这个命令输出结果中的63078d70-feb9-11e7-9812-dca90488bd22就是一个需要的UUID。

下面编写Python服务端的代码。创建一个名叫 **server.py** 的文件，在文件中输入下面的内容，代码中#后面的部分是注释。

```

# -*- coding: utf-8 -*-
from bluetooth import *

# 设置服务 UUID
uuid = "63078d70-feb9-11e7-9812-dca90488bd22"

# 创建服务端 Socket
bluetooth_socket = BluetoothSocket(RFCOMM)
bluetooth_socket.bind("", PORT_ANY)
bluetooth_socket.listen(1)

# 创建广播服务
advertise_service(
    bluetooth_socket,
    "Chat On Pi",
    service_id=uuid,
    service_classes=[uuid, SERIAL_PORT_CLASS],
    profiles=[SERIAL_PORT_PROFILE],
)

# 建立与客户端的连接
client, client_info = bluetooth_socket.accept()
print "客户连接: ", client_info

# 获取客户发送的内容
data = client.recv(1024)
print "客户发送了", data

# 向客户端发送内容
client.send("感谢你的信息")

# 关闭客户端连接
client.close()

# 关闭服务器连接
bluetooth_socket.close()

```

从代码中可以看出，Python编写的蓝牙服务端代码分为广播服务、建立连接、收发消息、关闭连接四个阶段。

广播服务用的是advertise_service方法。这个方法需要5个参数。

- 蓝牙Socket：请参考上面的代码来创建一个蓝牙Socket。
- 广播服务名字：一个自己起的名字，任意文本均可。
- 自己的唯一识别码service_id：这是我们生成的UUID。
- 服务类列表：这个服务支持的服务类（Service Class）列表。
- 服务型列表：这个服务支持的服务型（Service Profile）列表。

这里的 service 类和服务型将不做具体介绍。读者可以遵循代码中的设置。如果想要了解更多细节可以阅读Pybluez的文档。

42.3 蓝牙客户端程序

创建蓝牙客户端文件 *client.py*，并在里面输入下面的内容：

```

# -*- coding: utf-8 -*-
from bluetooth import *
import sys

# 获取服务
uuid = "63078d70-feb9-11e7-9812-dca90488bd22"
service_matches = find_service(uuid=uuid)

if len(service_matches) == 0:
    print("找不到对应的服务。")
    sys.exit(1)

first_match = service_matches[0]
port = first_match["port"]
name = first_match["name"]
host = first_match["host"]

print "找到蓝牙服务", first_match

# 创建客户端 Socket
socket = BluetoothSocket(RFCOMM)
socket.connect((host, port))

# 收发数据
socket.send("Hello Server!")
print socket.recv(1024)

# 关闭连接

socket.close()

```

同服务端一样，客户端的程序也分为四个步骤：获取服务、创建连接、收发数据、关闭连接。我们在获取服务时用到的方法是 `find_service`，只需要通过UUID就可以找到服务器端广播的服务。

42.4 服务端和客户端通信

首先，运行服务端。树莓派的蓝牙模块是默认不会被探索到的，我们需要在服务端运行下面两行命令让蓝牙服务器可以被客户端找到。

```
$sudo hciconfig hci0 piscan
$sudo hciconfig hci0 name 'My Chat Pi Server'
```

然后，运行服务端的Python程序：

```
$sudo pythonserver.py
```

需要注意的是，因为Python程序用到了蓝牙功能，所以需要有管理员权限才能执行这个脚本。

在客户端运行客户端脚本。

```
$sudo pythonclient.py
```

之后，我们会在服务端中看到输出。

```
客户连接: ('B8:27:EB:3B:2B:BA', 1)
客户发送了 Hello Server!
```

在客户端看到的输出如下所示。

```
找到蓝牙服务 {'protocol': 'RFCOMM', 'name': 'Chat On Pi', 'service-id':
'63078D70-FEB9-11E7-9812-DCA90488BD22', 'profiles': [('1101', 256)], 'service-
classes': [], 'host': 'B8:27:EB:7A:08:07', 'provider': None, 'port': 1,
'description': None}
感谢你的信息
```

42.5 实现文字聊天功能

42.4节的程序演示了最基本的数据传输，但是没有整合文字的输入功能，所以还无法通过这个程序来进行即时通信。下面修改这个Python脚本，使它能够从命令行中读取文字输入，从而实现聊天功能。

先来改动服务端，将服务端代码中收发数据的部分替换成下面的代码。

```
while True:
    # 获取客户发送的内容
    print "对方: ", client.recv(1024)

    q = raw_input("我: ")
    if q == "exit":
        break
    elif q:
        # 向客户端发送内容
        client.send(q)
```

这里的改动主要是：将发给客户端的内容变成了用raw_input指令从键盘读取。当用户输入exit之后会自动退出聊天。

类似地，将客户端收发数据的部分替换成下面的代码。

```
while True:
    q = raw_input("我: ")
    if q == "exit":
        break
    elif q:
        # 向客户端发送内容
        socket.send(q)

    # 获取客户发送的内容
    print "对方: ", socket.recv(1024)
```

在服务端和客户端运行这个程序后，双方就可以进行一人一句的聊天，运行效果如下所示。

```
我：你好吗？
对方：我很好。
我：今晚吃饭？
对方：好呀~
我：exit
（程序结束）
```


42.6 数据加密传输

由于42.5节的蓝牙聊天是明文传输的，因此别人可能窃取聊天内容。加密聊天内容可以让聊天信息变得安全。在介绍加密方法之前，要先了解一下常见的加密算法。我们可以将加密算法分为对称加密和非对称加密两大类。

对称加密算法通过密钥来加密或者解密数据。也就是说，当传递信息的双方都拥有同一个密钥时，就可以加密数据发给对方，并解密对方发来的加密数据。对称性算法很简单，但也有明显的缺点。一旦密钥泄露，得到密钥的人将可以解密所有信息。解决这个问题，就要确保不将密钥放在网络上传播，只将密钥与他人共享。于是，非对称性加密算法就诞生了。这个算法不再依赖于单一的密钥，而是每个人都有自己的私钥和公钥，以及对方的公钥。假如A要给B发送一个消息，A将会使用B的公钥加密这个信息，B收到A发来的消息后，用自己的私钥来解密信息。也就是说，对于一组私钥和公钥，任何人都可以用公钥来加密信息，但只有持有私钥的人可以解密被公钥加密的信息。

这样在一段聊天开始时，只要让聊天的双方将自己的公钥发给对方，用对方的公钥来加密要发送给对方的数据，再用自己的私钥来解密收到的数据，就可以实现安全通信了。这里将使用一个名叫Python-RSA的模块。这个模块提供了纯Python语言的RSA算法的实现。RSA是最常用的非对称性算法之一。

可以用pip命令安装Python-RSA模块。

```
$sudo pip install rsa
```

创建一个叫作security的Python模块。方法是新建一个名为 *security.py* 的文件，将下面的脚本输入进去。

```
# -*- coding: utf-8 -*-
import rsa

pubkey, privkey = rsa.newkeys(512)

def get_public_key():
    return pubkey.save_pkcs1()

def encrypt(message, keydata):
    key = rsa.PublicKey.load_pkcs1(keydata)
    return rsa.encrypt(message, key)

def decrypt(message):
    return rsa.decrypt(message, privkey)
```

这个模块将会自动生成一对RSA的公钥和密钥，并提供三个函数。

- get_public_key：获取生成的公钥使用PEM文本格式。
- encrypt：使用keydata加密信息message。Keydata是PEM的文本格式。
- decrypt：使用自己的密钥解密被加密的信息message。

实现完加密模块后，便可以将它用到聊天系统中。首先，在客户端和服务端的程序上引用这个模块，方法是加入下面这一行代码。

```
import security
```

然后，将下面的代码分别放置在客户端和服务端收发信息的while循环前面，放在服务端的代码如下所示。

```
my_public_key = security.get_public_key()
client.send(my_public_key)
client_public_key = client.recv(1024)
```

放在客户端的代码如下所示。

```
my_public_key = security.get_public_key()
socket.send(my_public_key)
server_public_key = socket.recv(1024)
```

这两段代码的作用是互相交换公钥。这样就可以在给对方发送消息之前加密信息。

最后，修改收发消息的函数，使它们使用加密模块的算法。例如，将`socket.send(q)`替换成`socket.send(security.encrypt(q,server_public_key))`原始的信息 `q` 就被加密了，而将 `socket.recv(1024)` 替换成 `security.decrypt(socket.recv(1024))`就可以解密加密的信息了。完整的服务端脚本如下所示。

```
# -*- coding: utf-8 -*-
from bluetooth import *
import security

# 设置服务 UUID
uuid = "63078d70-feb9-11e7-9812-dca90488bd22"

# 创建服务端 Socket
bluetooth_socket = BluetoothSocket(RFCOMM)
bluetooth_socket.bind(("", PORT_ANY))
bluetooth_socket.listen(1)

# 创建广播服务
advertise_service(
    bluetooth_socket,
    "Chat On Pi",
    service_id=uuid,
```

```

        service_classes=[uuid, SERIAL_PORT_CLASS],
        profiles=[SERIAL_PORT_PROFILE],
    )

    # 获取客户端连接
    client, client_info = bluetooth_socket.accept()
    print "客户连接: ", client_info

    my_public_key = security.get_public_key()
    client.send(my_public_key)
    client_public_key = client.recv(1024)

    while True:
        # 获取客户发送的内容
        print "对方: ", security.decrypt(client.recv(1024))

        q = raw_input("我: ")
        if q == "exit":
            break
        elif q:
            # 向客户端发送内容
            client.send(security.encrypt(q, client_public_key))

    # 关闭客户端连接
    client.close()

    # 关闭服务器连接
    bluetooth_socket.close()

```

完整的客户端脚本如下:

```

# -*- coding: utf-8 -*-
from bluetooth import *
import sys
import security

# 获取服务
uuid = "63078d70-feb9-11e7-9812-dca90488bd22"
service_matches = find_service(uuid=uuid)

if len(service_matches) == 0:
    print("找不到对应的服务。")
    sys.exit(1)

first_match = service_matches[0]

```

```
port = first_match["port"]
name = first_match["name"]
host = first_match["host"]

print "找到蓝牙服务", first_match

# 创建客户端 Socket
socket = BluetoothSocket(RFCOMM)
socket.connect((host, port))

my_public_key = security.get_public_key()
socket.send(my_public_key)
server_public_key = socket.recv(1024)

# 收发数据
while True:
    q = raw_input("我: ")
    if q == "exit":
        break
    elif q:
        # 向客户端发送内容
        socket.send(security.encrypt(q, server_public_key))

    # 获取客户发送的内容
    print "对方: ", security.decrypt(socket.recv(1024))

# 关闭连接
socket.close()
```

第43章 制作一个Shell

Shell是Linux操作系统上的命令解释器，它是最传统的Linux用户交互方式。一个Linux Shell通常具有以下功能。

- 读取用户输入指令。
- 浏览文件系统。
- 执行可执行程序。
- 处理程序输出和异常。

我们已经学习了功能强大的bash。本章用C语言来编写一个基本的Shell。遵循常见Linux的Shell程序的命名方式，我们将Shell程序命名为crash。

43.1 配置项目

一个Shell程序还是相当复杂的。为了方便未来的开发，我们需要安排源代码项目的结构。这里将使用make来管理源代码和编译程序。make工具将会读取一个名为 *Makefile* 的文件来决定如何编译当前的项目。

1.Makefile

Makefile 文件是make的配置文件，它决定了make工具将按照怎样的结构顺序来编译当前项目。*crash* 项目的结构安排如下。

```
根目录 crash
— README, 程序的介绍
— Makefile, 项目配置
— src, 源代码
— obj, 编译产生的目标文件
— bin, 编译产生的可执行文件
```

根据这个设计，*Makefile* 将会是下面的样子^[1]。

```
# 编译好的可执行程序的名称
TARGET = crash

# 编译器，我们这里使用 gcc
CC = gcc
# 编译器标志 (Flag)
CFLAGS = -std=c99 -Wall -I. -g

# 连接器，这里我们继续使用 gcc
LINKER = gcc
# 连接器标志
LFLAGS = -Wall -I. -lm

# 源代码目录
SRCDIR = src
# 目标文件目录
OBJDIR = obj
# 二进制可执行文件目录
BINDIR = bin

# 所有源代码文件
SOURCES := $(shell find $(SRCDIR) -type f -name '*.c')
# 所有头文件
INCLUDES := $(shell find $(SRCDIR) -type f -name '*.h')
# 所有目标文件
OBJECTS := $(SOURCES:$(SRCDIR)/%.c=$(OBJDIR)/%.o)

# 连接规则
$(BINDIR)/$(TARGET): $(OBJECTS)
    @mkdir -p $(BINDIR)
    @$(LINKER) $(OBJECTS) $(LFLAGS) -o $@
    @echo "Linking complete!"

# 编译规则
$(OBJECTS): $(OBJDIR)/%.o : $(SRCDIR)/%.c
    @mkdir -p $(dir $@)
    @$(CC) $(CFLAGS) -c $< -o $@
    @echo "Compiled "$<" successfully!"

# 清理编译文件
```

```
.PHONY: clean
clean:
    @rm -rf $(BINDIR) $(OBJDIR)
    @echo "Cleanup complete!"
```

2.创建源代码文件

源代码是编写应用程序的关键。为了测试 *Makefile* 文件是否编写成功，我们在 *src* 目录下创建第一个源代码文件 *main.c*，内容如下。

```
#include <stdio.h>

//程序入口
int main()
{
    //输出 Hello World
    printf("Hello World.\n");

    //结束程序
    return 0;
}
```

使用 *make* 命令测试编译：

```
$make
Compiled src/main.c successfully!
Linking complete!
```

编译成功后，查看 *obj* 文件夹中的内容会看到里面有一个名为 *main.o* 的目标文件，而目录 *bin* 中产生了可执行文件 *crash*。

直接运行 *crash* 文件：

```
$/bin/crash
Hello World.
```

43.2 输入输出设置

由于Shell程序涉及很多用户交互，它的输入输出行为和普通的应用程序有较大区别。因此，我们需要修改命令行行为的输入输出标志TIO（Terminal IO）。这里设置的标志是ICANON和ECHO。在 *src* 文件夹中新建一个文件 *tio.c* 来负责修改输入输出标志：

```
#include <unistd.h>
#include <termios.h>
#include <signal.h>

struct termios backup_tio_settings;

void backup_tio()
{
    tcgetattr(STDIN_FILENO, &backup_tio_settings);
}

void set_tio()
{
    struct termios new_tio;

    //创建新的设置
    new_tio = backup_tio_settings;
    new_tio.c_lflag &= (~ICANON & ~ECHO);

    //设置 TIO
    tcsetattr(STDIN_FILENO, TCSANOW, &new_tio);
}

void restore_tio()
{
    tcsetattr(STDIN_FILENO, TCSANOW, &backup_tio_settings);
}
```

文件里定义了三个函数。

- backup_tio，保存系统默认的TIO设置。
- set_tio，将TIO设置成我们希望的值。
- restore_tio，将TIO恢复成系统默认的TIO设置。

此外，我们需要给上面的C语言程序编写一个头文件，来包含声明所有的函数。针对 *tio.c* 创建头文件 *tio.h* 并输入下面的内容来声明这三个函数。

```
void backup_tio();  
void set_tio();  
void restore_tio();
```

43.3 初步的Shell

我们可以编写Shell的主程序main.c。因为我们尚未编写好Shell的核心功能，所以可以暂且用shell.h中的一个input_loop函数来代表核心功能，以及一个sigint_handler来捕捉SIGINT信号。

```

#include <signal.h>
#include <stdio.h>

#include "shell.h"
#include "tio.h"

//程序入口
int main()
{
    //禁用输出缓存。因为默认 Linux 的程序输出会有缓存，这样输出的内容可能不会即时显示在屏幕上
    //因为 Shell 是呼叫程序，所以这里要禁用程序输出的缓存
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    //这里还需要修改 Terminal I/O 的设置
    backup_tio();
    set_tio();

    //绑定快捷键 Ctrl+C 发送的 SIGINT 事件
    signal(SIGINT, sigint_handler);

    //启动 Shell 的输入循环
    int return_value = input_loop();

    //恢复之前的 TIO 设置
    restore_tio();

    //结束程序
    return return_value;
}

```

在这段程序中，我们禁用输出缓存，从而让程序输出立即显示在用户命令行中。如果不禁用输出缓存，那么程序输出的内容可能只有在积累到一定量之后才能显示出来。禁用输出缓存的方法如下所示。

```

setbuf(stdout, NULL);
setbuf(stderr, NULL);

```

此外，这段程序还可以捕捉信号。

```
signal(SIGINT, sigint_handler);
```

其中sigint_handler是一个会在下文编写的信号处理函数。

43.4 文字颜色与其他配置

Shell输出的文字可以是多种颜色的。虽然这不是Shell的必要特性，但是确实可以提高用户体验。为了方便使用，我们将常用的颜色常量放在一个单独的 *color.h* 文件里。

```
#define ANSI_COLOR_RED      "\x1b[31m"  
#define ANSI_COLOR_GREEN   "\x1b[32m"  
#define ANSI_COLOR_YELLOW  "\x1b[33m"  
#define ANSI_COLOR_BLUE    "\x1b[34m"  
#define ANSI_COLOR_MAGENTA "\x1b[35m"  
#define ANSI_COLOR_CYAN    "\x1b[36m"  
#define ANSI_COLOR_RESET   "\x1b[0m"
```

可以看到，以.h为后缀的头文件可以用 #define来定义配置变量，格式为：

```
#define <变量名> <变量值>
```

C程序中所有出现的变量名将替换为变量值。

通常，输出一段文字的方法是：

```
printf("我要输出的文字");
```

假如要把“我”字变成蓝色，那么只需要将代码修改为：

```
printf(ANSI_COLOR_BLUE "我" ANSI_COLOR_RESET "要输出的文字");
```

这样“我”字就变成蓝色了。

此外，使用一个单独的 *config.h* 文件来保存其他配置。crash将用到下面四个变量，读者可以直接使用下面定义的值：

```
#define INPUT_BUFFER_SIZE 1024
#define MAX_PATH_LENGTH 1024
#define MAX_ARGUMENT_NUMBER 128
#define MAX_PATH_NUMBER 128
```

43.5 部分Shell 功能

用C语言编写Shell下的常见功能。

1.改变工作目录

cd用于改变工作目录，一般由Shell提供。为了让Shell有改变工作目录的功能，可以先编写一个名为command_cd的函数，这个函数的声明是：

```
int command_cd(char *path);
```

改变工作目录需要用到的函数是chdir(char *)，这个函数会带有一个参数，是希望改变到的目标目录。这个函数的返回值是一个整数，如果成功，则返回0，如果错误，则返回非0整数。

需要特殊处理的地方是，在Shell中可以使用~来表示 *Home* 目录，比如用户可以用下面的命令来进入 *Home* 目录中的 *Documents* 文件夹。

```
$cd ~/Documents
```

我们需要判断用户输入的路径的第一个字母是不是~，如果是，则需要用实际的Home路径替换它。

获取Home实际路径的方法略微有些复杂。为了兼容不同系统，我们需要先检查环境变量中的HOME参数是否为空。如果HOME不为空，则使用这个值，否则使用getpwuid(getuid())->pw_dir的结果。获取 *Home* 目录的代码如下：

```

char *home_path;

if ((home_path = getenv("HOME")) == NULL) {
    Home_path = getpwuid(getuid())->pw_dir;
}

```

结合以上的内容，cd命令的代码如下：

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <pwd.h>

#include "../config.h"

int command_cd(char *path)
{
    if (path[0] == '~') {
        char *home_path;

        if ((home_path = getenv("HOME")) == NULL) {
            Home_path = getpwuid(getuid())->pw_dir;
        }

        char new_path[MAX_PATH_LENGTH];
        strcpy(new_path, Home_path);
        strcat(new_path, path + 1);
        path = new_path;
    }

    int error = chdir(path);
    if (error) {
        printf("Unable to change directory to \"%s\".\n", path);
    }
    return error;
}

```

将cd命令的代码保存到 *src/commands/cd.c* 文件中，对应的头文件在同目录中的 *cd.h* 文件中。头文件只需要包含command_cd的函数声明即可。

2.列出当前目录所有文件

ls是另一个常见的命令。用opendir函数和readdir函数来获得当前工作目录中的所有文件，代码如下：

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
```

int skIP= 2; //列表中前两个项目是.和..分别代表当前目录和父目录，我们跳过

```
int command_ls()
{
    DIR *dp;
    struct dirent *ep;

    dp = opendir("./");
    if (dp != NULL)
    {
        int count = 0;
```

```

        while ((ep = readdir(dp)))
        {
            count++;
            if (count > skip)
            {
                puts(ep->d_name);
            }
        }
        closedir(dp);
    }
    else
        perror("Couldn't open the directory");

    return 0;
}

```

这段代码放在 `src/commands/ls.c` 文件里，头文件是 `src/commands/ls.h`，其中包含了 `command_ls` 的函数声明。

3. 执行可执行程序

执行可执行程序用到的是 `execvp` 函数。这个函数自带两个参数，其声明如下：

```
int execvp(const char *file, char *const argv[])
```

第一个参数是可执行文件的目录，第二个参数是一个字符串的列表，作为执行进程时的参数列表。

需要注意的是，`execvp` 会直接在当前进程执行这个可执行程序。这样做的后果是，Shell 将失去对当前进程的控制，例如无法中断它的执行。为了解决这个问题，我们使用 `fork` 方法创建了一个子进程，在子进程中使用 `execvp`，而父进程将等待子进程的完成，等待的代码如下：


```

do
{
    waitpid(pid, &status, WUNTRACED);
} while (!WIFEXITED(status) && !WIFSIGNALED(status));

```

完整的 launch_process 代码如下：

```

int launch_process(char **args)
{
    pid_t pid = fork();
    if (pid == 0) //子进程
    {

        //恢复正常的 TIO 设置给子进程
        restore_tio();
        if (execvp(args[0], args) == -1)
        {
            perror("lanuch");
        }
        exit(EXIT_SUCCESS);
    }
    else if (pid < 0) //无法创建子进程, 打印错误
    {
        perror("lanuch");
    }
    else //父进程
    {
        int status;
        is_running = 1;
        printf("Start child process %s at PID %d...\n", args[0], pid);
        do
        {
            waitpid(pid, &status, WUNTRACED);
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        is_running = 0;
        set_tio();
        return status;
    }

    return 1;
}

```

我们在上面的程序中增加了一个小功能，就是用is_running变量记录了当前是否有子进程被执行。这将会被Shell主程序用到。

除了launch_process函数之外，还需要一个函数在当前系统PATH环境变量中找到对应的可执行程序。这里实现了一个函数char *get_executable_path(char *command)。它的输入是一个程序的名字，输出是完整的路径。get_executable_path的完整代码如下：

```
char *get_executable_path(char *command)
{
    char* env_path = getenv("PATH");
    int path_length = 1;
    char *path_array[MAX_PATH_NUMBER];
    int current_path_index = 0;
    int current_position = 0;
    path_array[current_path_index] = malloc(MAX_PATH_LENGTH);
```

```

for (int i = 0; i < strlen(env_path); i++) {
    if (env_path[i] == ':') {
        path_array[current_path_index][current_position] = '\\0';
        current_position = 0;
        current_path_index ++;
        path_array[current_path_index] = malloc(MAX_PATH_LENGTH);
        continue;
    }
    path_array[current_path_index][current_position] = env_path[i];
    current_position ++;
}
path_array[current_path_index][current_position] = 0;
path_array[current_path_index + 1] = 0;

path_length = current_path_index + 1;

for (int i = 0; i < path_length; i++)
{
    char *path = path_array[i];
    char *executable_path = (char *)malloc(MAX_PATH_LENGTH);
    strcpy(executable_path, path);
    strcat(executable_path, "/");
    strcat(executable_path, command);
    if (is_regular_file(executable_path))
    {
        //文件存在
        return executable_path;
    }
    else
    {
        //文件不存在
        free(executable_path);
    }
}

//释放 path_array 使用的内存
for (int i = 0; i < path_length; i++) {
    free(path_array[i]);
}
return NULL;
}

```

上面代码放在了 *src/launch.c* 文件里，对应的头文件是 *src/launch.h*

。

下面是完整的 *launch.c* 文件。

```

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

#include "config.h"
#include "tio.h"

bool is_running = 0;

int launch_process(char **args)
{
    pid_t pid = fork();
    if (pid == 0) //子进程
    {
        //恢复正常的 TIO 设置给子进程
        restore_tio();
        if (execvp(args[0], args) == -1)
        {
            perror("lanuch");
        }
        exit(EXIT_SUCCESS);
    }
    else if (pid < 0) //无法创建子进程, 打印错误
    {
        perror("lanuch");
    }
    else //父进程
    {
        int status;
        is_running = 1;
        printf("Start child process %s at PID %d...\n", args[0], pid);
        do
        {
            waitpid(pid, &status, WUNTRACED);
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        is_running = 0;
        set_tio();
        return status;
    }
}

```

```

    return 1;
}

bool is_child_process_running() {
    return is_running;
}

int is_regular_file(const char *path)
{
    struct stat path_stat;
    stat(path, &path_stat);
    return S_ISREG(path_stat.st_mode);
}

char *get_executable_path(char *command)
{
    char* env_path = getenv("PATH");
    int path_length = 1;
    char *path_array[MAX_PATH_NUMBER];
    int current_path_index = 0;
    int current_position = 0;
    path_array[current_path_index] = malloc(MAX_PATH_LENGTH);
    for (int i = 0; i < strlen(env_path); i++) {
        if (env_path[i] == ':') {
            path_array[current_path_index][current_position] = '\0';
            current_position = 0;
            current_path_index ++;
            path_array[current_path_index] = malloc(MAX_PATH_LENGTH);
            continue;
        }
        path_array[current_path_index][current_position] = env_path[i];
        current_position ++;
    }
    path_array[current_path_index][current_position] = 0;
    path_array[current_path_index + 1] = 0;

    path_length = current_path_index + 1;

    for (int i = 0; i < path_length; i++)
    {
        char *path = path_array[i];
        char *executable_path = (char *)malloc(MAX_PATH_LENGTH);
        strcpy(executable_path, path);
        strcat(executable_path, "/");
        strcat(executable_path, command);
    }
}

```

```
        if (is_regular_file(executable_path))
        {
            //文件存在
            return executable_path;
        }
        else
        {
            //文件不存在
            free(executable_path);
        }
    }

    //释放 path_array 使用的内存
    for (int i = 0; i < path_length; i++) {
        free(path_array[i]);
    }
    return NULL;
}
```

43.6 Shell主程序

掌握了43.5节的Shell功能后，我们开始编写shell.c代码，将各个功能拼接在一起。Shell主程序的主要任务有控制输入输出、调用对应函数。代码全文如下：

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#include "shell.h"
#include "color.h"
#include "launch.h"
#include "commands/ls.h"
#include "commands/cd.h"

//输入缓存, 当前行输入的内容
char input_buffer[INPUT_BUFFER_SIZE];
//当前命令, 第一个输入的内容
char current_command[INPUT_BUFFER_SIZE];
//当前工作路径
char current_working_directory[MAX_PATH_LENGTH];
//当前参数
```



```

char *current_arguments[MAX_ARGUMENT_NUMBER];

int return_value = 0;
bool exited = false;
int position = 0;

/**
 * 处理当前输入缓存里的输入。将输入信息写入 current_command 和 current_arguments 中
 */
void process_input()
{
    int argument_index = 0;
    int current_position = 0;
    current_arguments[0] = malloc(MAX_PATH_LENGTH);
    for (int i = 0; i < strlen(input_buffer); i++)
    {
        if (input_buffer[i] == ' ')
        {
            //终止当前的命令
            current_arguments[argument_index][current_position] = 0;
            argument_index++;
            current_position = 0;
            current_arguments[argument_index] = malloc(MAX_PATH_LENGTH);
            continue;
        }
        current_arguments[argument_index][current_position] = input_buffer[i];
        current_position++;
    }
    //终止当前的命令
    current_arguments[argument_index][current_position] = 0;
    current_arguments[argument_index + 1] = 0;

    strcpy(current_command, current_arguments[0]);
}

void free_input() {

}

/**
 * 检查所给命令是否和输入命令相同
 * @param command 需要查询的命令
 */
bool check_command(char *command)
{

```

```

    return strcmp(current_command, command) == 0;
}

/**
 * 检查输入的命令是否在 PATH 中有对应的程序
 */
bool exist_in_path()
{
    return get_executable_path(current_command) != NULL;
}

/**
 * 检查指定字符串是否是输入命令的前缀
 */
bool start_with(char *prefix)
{
    return strncmp(prefix, current_command, strlen(prefix)) == 0;
}

/**
 * 获取当前文件夹名称
 */
void get_current_folder(char *output)
{
    int last_slash = 0;
    int cwd_length = strlen(current_working_directory);
    for (int i = 0; i < cwd_length; i++)
    {
        if (current_working_directory[i] == '/')
        {
            last_slash = i;
        }
    }
    for (int i = last_slash + 1; i < cwd_length; i++)
    {
        output[i - last_slash - 1] = current_working_directory[i];
    }
    output[cwd_length - last_slash - 1] = 0;
}

/**
 * 准备一行新的 s 输入
 */
void restart_input()
{

```

```

        position = 0;
        getcwd(current_working_directory, sizeof(current_working_directory));
        char current_folder[MAX_PATH_LENGTH];
        get_current_folder(current_folder);
        printf(ANSI_COLOR_BLUE "%s " ANSI_COLOR_YELLOW "> " ANSI_COLOR_RESET,
current_folder);
    }

```

```

/**

```

```

 * 处理按键

```

```

 * @param c 按键代码

```

```

 */

```

```

bool process_key(unsigned char c)

```

```

{

```

```

    if (c == '\n') //Newline 回车

```

```

    {

```

```

        printf("\n");

```

```

        return true;

```

```

    }

```

```

    else if (c == 127 || c == 8) //Delete 键或者 Backspace 键

```

```

    {

```

```

        if (position > 0)

```

```

        {

```

```

            printf("\b \b");

```

```

            position--;

```

```

        }

```

```

    }

```

```

    else if (c == '\t') //Tab 键

```

```

    {

```

```

    }

```

```

    else if (c == 27) //Escape 键

```

```

    {

```

```

        char next = getchar();

```

```

        if (next == 91)

```

```

        {

```

```

            switch (getchar())

```

```

            {

```

```

                case 65:

```

```

                    break;

```

```

            }

```

```

        }

```

```

        else

```

```

        {

```

```

            return process_key(next);

```

```

        }

```

```

    }
    else //其他按键
    {
        printf("%c", c);
        //加进输入缓存里
        input_buffer[position] = c;
        position++;
    }
    return process_key(getchar());
}

/**
 * Shell 程序主循环
 */
void line_loop()
{
    restart_input();

    while (true)
    {
        if (process_key(getchar()))
        {
            break;
        }
    }

    input_buffer[position] = 0;
    process_input();

    if (check_command("exit"))
    {
        exited = true;
        return_value = 0;
    }
    else if (check_command("ls"))
    {
        command_ls();
    }
    else if (check_command("cd"))
    {
        command_cd(current_arguments[1]);
    }
    else if (check_command("pwd"))
    {
        printf("%s\n", current_working_directory);
    }
}

```

```

    }
    else if (check_command("clear"))
    {
        printf("\e[1;1H\e[2J");
    }
    else if (exist_in_path() || start_with("./"))
    {
        if (!start_with("./")) {
            //替换完整路径
            current_arguments[0] = get_executable_path(current_command);
        }
        launch_process(current_arguments);
    }
    else
    {
        if (strlen(current_command))
        {
            printf("Command \"%s\" is not found.\n", current_command);
        }
    }
    free_input();
}

/**
 * 输入循环
 */
int input_loop()
{
    while (!exited)
    {
        line_loop();
    }

    printf("Bye.\n\n");

    return return_value;
}

/**
 * 响应 SIGINT 信号
 */
void sigint_handler()
{
    if (is_child_process_running()) {

```

```
    } else {  
        printf("\nPlease use \"exit\" command to exit the shell.\n");  
        printf("\n");  
        restart_input();  
    }  
}
```

现在，该检验本章的成果了，运行 ***./bin/crash*** 文件可以进入我们编写的Shell程序。这里可以体验一下它的功能，比如获取当前活动路径：

```
Crash > pwd  
/Users/dreamrunner/Projects/Crash
```

改变目录：

```
Crash > cd src
```

列出当前目录文件：

```
src > ls  
color.h  
commands  
config.h  
launch.c  
launch.h  
main.c  
shell.c  
shell.h  
tio.c  
tio.h
```

此外，还可以像bash那样使用快捷键Ctrl+C发出SIGINT信号，用exit命令退出Shell：

```
Crash > exit  
Bye.
```

本章通过实现一个简单的Shell深入体验了一次Linux应用程序的开发。

[1] 在 *Makefile* 中，#后面的文字是注释。

第44章 人工智能

人工智能（AI，Artificial Intelligence）是近年来热门的话题。人工智能让机器具备人类的智慧。**深度学习**（Deep Learning）是人工智能的重要分支。深度学习通过多个层次的处理来对数据进行高层抽象。深度学习在图像识别、语音识别、生物信息等领域取得惊人的突破。而在围棋项目中，AlphaGo更是轻松击败人类顶尖选手，引发了公众对人工智能的大讨论。本章将让树莓派也拥有深度学习的“超能力”。

44.1 树莓派的准备

本章将在树莓派上运行深度学习程序，并结合树莓派的摄像头，最终做到**实时物品识别**（Real-time Object Detection）。具体的深度学习算法是YOLO（You Only Look Once）。YOLO的主要发明人是约瑟夫·雷德蒙（Joseph Redmon），它以处理速度超快著称，正适合树莓派这样的微型电脑。我们使用的YOLO程序是基于TensorFlow的DarkFlow。此外，还需要一个摄像头来给树莓派输入画面。

因为Darkflow程序依赖于Python 3、TensorFlow和OpenCV 3，所以要在树莓派上安装这些程序。Raspbian默认安装的是Python 2.7版本。我们要确保树莓派上安装了Python 3和pip3。方法如下：

```
$sudo apt-getinstall python 3-pip python 3-dev
$sudo pip3 install --upgrade pip
```

安装TensorFlow。TensorFlow是谷歌开发的一个机器学习开源工具集。由于TensorFlow官方并没有发布适合树莓派的发布包，这里使用了一个第三方制作的适合树莓派的TensorFlow安装包^[1]，脚本如下：

```
$wget https://github.com/samjabrahams/tensorflow-on-raspberry-
pi/releases/download/v1.1.0/tensorflow-1.1.0-cp34-cp34m-linux_armv7l.whl
$sudo pip3 install tensorflow-1.1.0-cp34-cp34m-linux_armv7l.whl
```


安装OpenCV 3。OpenCV是一个经典的计算机视觉代码库，提供了很多必备的图像处理工具。在树莓派上安装OpenCV 3相对烦琐。不过只要一步一步安装，相信大家都可以安装成功的。整个安装过程耗时大约两个小时。

第一步，更新apt-get的软件包和系统已安装的软件：

```
$sudoapt-getupdate && sudoapt-getupgrade
```

第二步，安装包括cmake在内的开发者工具，这是因为我们需要从源代码安装。

```
$sudoapt-getinstall build-essential cmake pkg-config
```

第三步，安装一些依赖的库：

```
$sudoapt-getinstall libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev  
$sudoapt-getinstall libavcodec-dev libavformat-dev libswscale-dev libv4l-dev  
$sudoapt-getinstall libxvidcore-dev libx264-dev  
$sudoapt-getinstall libatlas-base-dev gfortran
```

第四步，下载OpenCV 3的源代码，这里需要下载两个压缩包opencv.zip和opencv_contrib.zip。前者是OpenCV核心的源代码，后者是一些附加源码包。

```
$cd ~  
$wget-0 opencv.zip https://github.com/Itseez/opencv/archive/3.3.0.zip  
$unzip opencv.zip  
$wget-0  
opencv_contrib.zip https://github.com/Itseez/opencv_contrib/archive/3.3.0.zip  
$unzip opencv_contrib.zip
```

第五步，使用cmake来创建目标。

```
$cd ~/opencv-3.3.0/  
$mkdir build  
$cd build  
$cmake -D CMAKE_BUILD_TYPE=RELEASE \  
-D CMAKE_INSTALL_PREFIX=/usr/local \  
-D INSTALL_PYTHON_EXAMPLES=ON \  
-D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-3.3.0/modules \  
-D BUILD_EXAMPLES=ON ..
```

第六步，编译源代码。这一步是整个安装过程中最耗时的步骤，总共需要大约一个半小时。

```
$make -j4
```

将编译好的二进制文件放进系统目录，完成安装。

```
$sudo make install  
$sudo ldconfig
```

测试一下安装是否成功，检查方法是在Python 3的import cv2包中输出cv2包：

```
$python 3  
>>>import cv2  
>>>cv2  
  
<module 'cv2' from '/usr/local/lib/Python 3.4/dist-packages/cv2.cpython-34m.so'>
```

如果可以看到类似上面的结果，即可以正常引入cv2包，并且看到它来自一个 **so** 文件，这就说明OpenCV 3已经安装成功了。

安装DarkFlow。首先下载Darkflow源代码：

```
$git clone git@github.com:thtrieu/darkflow.git
```

然后使用inplace的方式安装Darkflow。

```
$cd darkflow
$Python 3 setup.py build_ext -inplace

running build_ext
copying build/lib.linux-armv7l-3.4/darkflow/cython_utils/nms.cpython-34m.so ->
darkflow/cython_utils
copying build/lib.linux-armv7l-
3.4/darkflow/cython_utils/cy_yolo2_findboxes.cpython-34m.so ->
darkflow/cython_utils
copying build/lib.linux-armv7l-
3.4/darkflow/cython_utils/cy_yolo_findboxes.cpython-34m.so -> darkflow/cython_utils
```

如果看到上面的输出说明安装成功了。安装成功后，为了方便，我们也可以在全局安装Darkflow，在同一个文件夹下执行命令：

```
$sudo pip install -e .
```

全局安装好后，就可以在任何目录下编写Python脚本，并在脚本中使用Darkflow了。

44.2 YOLO识别

在使用深度学习算法之前，通常需要用真实的数据来训练该算法。然而，训练一个深度学习的系统需要大量的训练数据。幸好YOLO算法已经有现成的训练数据。下载一个已经训练好的数据集。

```
$wget https://raw.githubusercontent.com/pjreddie/darknet/master/cfg/tiny-yolo-
voc.cfg
$wget https://pjreddie.com/media/files/tiny-yolo-voc.weights
```

这里下载了两个文件，第一个是配置文件 *tiny-yolo-voc.cfg*，第二个是训练结果 *tiny-yolovoc.weights*。

用YOLO检测一张照片。首先，创建一个名叫detect.py的程序，代码如下：

```

# 第一部分
from darkflow.net.build import TFNet
import cv2
import time
import PIL
import numpy

# 第二部分
options = {"model": "tiny-yolo-voc.cfg", "load": "tiny-yolo-voc.weights",
"threshold": 0.1}

tfnet = TFNet(options)

# 第三部分
while True:
    curr_img = PIL.Image.open(open("image.jpg", "rb"))
    curr_img_cv2 = cv2.cvtColor(numpy.array(curr_img), cv2.COLOR_RGB2BGR)
    result = tfnet.return_predict(curr_img_cv2)
    print(result)
    time.sleep(5)

```

这段Python脚本分为三部分。第一部分引入了一些这个脚本需要的Python库。第二部分创建一个TFNet深度学习网络，这一部分我们配置了三个选项。

- model：之前下载的 *cfg* 文件。
- load：之前下载的 *weights* 文件。
- threshold：检测时需要用到的一个阈值，暂时设置成0.1。

第三部分检测部分脚本。这个脚本被放在了一个while循环中，每5秒执行一次。这个脚本会读取一个名为 *image.jpg* 的文件，并使用tfnet.return_predict方法来检测图片中可能出现的物体。

脚本输入完后，可以用Python 3来执行这个脚本：

```
$Python 3 detect.py
```

```
Parsing tiny-yolo-voc.cfg
Loading tiny-yolo-voc.weights ...
Successfully identified 63471556 bytes
Finished in 0.0631105899810791s
Model has a VOC model name, loading VOC labels.
```

```
Building net ...
```

Source	Train?	Layer description	Output size
		input	(?, 416, 416, 3)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 416, 416, 16)
Load	Yep!	maxp 2x2p0_2	(?, 208, 208, 16)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 208, 208, 32)
Load	Yep!	maxp 2x2p0_2	(?, 104, 104, 32)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 104, 104, 64)
Load	Yep!	maxp 2x2p0_2	(?, 52, 52, 64)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 52, 52, 128)
Load	Yep!	maxp 2x2p0_2	(?, 26, 26, 128)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 26, 26, 256)
Load	Yep!	maxp 2x2p0_2	(?, 13, 13, 256)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 13, 13, 512)
Load	Yep!	maxp 2x2p0_1	(?, 13, 13, 512)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 13, 13, 1024)
Load	Yep!	conv 3x3p1_1 +bnorm leaky	(?, 13, 13, 1024)
Load	Yep!	conv 1x1p0_1 linear	(?, 13, 13, 125)

```
Running entirely on CPU
Finished in 13.229841947555542s
```

```
[{'bottomright': {'y': 432, 'x': 119}, 'confidence': 0.2320941, 'label':
'bicycle', 'topleft': {'y': 198, 'x': 24}}, {'bottomright': {'y': 314, 'x': 126},
'confidence': 0.2504689, 'label': 'bird', 'topleft': {'y': 178, 'x': 13}}]
```

运行到这里，就可以看到程序识别出两个目标。这个结果被 `print(result)` 命令打印出来了。

- `label`：物品的名称。
- `confidence`：信心指数，越高说明深度学习越认为这个物品就是它。
- `topleft`和`bottomright`：物品右上角和左下角的坐标，单位是像素。

上面是用一张图片做测试。在应用的时候，可以直接用YOLO来识别摄像头实时拍到的画面。第13章介绍过使用树莓派摄像头拍照片的方法：

```
import subprocess
subprocess.call(["raspistill", "-o", "filename.jpg"])
```

那么我们只需要简单修改之前的脚本，就可以把固定的“image.jpg”替换成摄像头刚刚拍摄的照片。

修改后的程序代码如下：

```
from darkflow.net.build import TFNet
import cv2
import time
import PIL
import numpy
import subprocess

options = {"model": "tiny-yolo-voc.cfg", "load": "tiny-yolo-voc.weights",
"threshold": 0.1}

tfnet = TFNet(options)

count = 0
while True:
    filename = "%s.jpg" % count
    subprocess.call(["raspistill", "-o", filename])
    curr_img = PIL.Image.open(open(filename, "rb"))
    curr_img_cv2 = cv2.cvtColor(numpy.array(curr_img), cv2.COLOR_RGB2BGR)
    result = tfnet.return_predict(curr_img_cv2)
    print(result)
    time.sleep(5)
    count += 1
```

44.3 图形化显示结果

现在的程序只能把预测结果打印在命令行上，我们可以用Python编写一个小脚本，将摄像头拍摄的画面和YOLO识别结果显示在屏幕上。

首先，安装imagemagick来显示图片：

```
$sudoapt-getinstall imagemagick
```

然后，修改之前的Python脚本，在上面加上一小段代码，将识别出的物品标记出来：

```
from darkflow.net.build import TFNet
import cv2

import time
import PIL
import numpy
import subprocess

options = {"model": "tiny-yolo-voc.cfg", "load": "tiny-yolo-voc.weights",
"threshold": 0.1}

tfnet = TFNet(options)

count = 0
while True:
    filename = "%s.jpg" % count
    subprocess.call(["raspistill", "-o", filename])
    curr_img = PIL.Image.open(open(filename, "rb"))
    curr_img_cv2 = cv2.cvtColor(numpy.array(curr_img), cv2.COLOR_RGB2BGR)
    result = tfnet.return_predict(curr_img_cv2)
    print(result)
    time.sleep(5)
    count += 1
```

再次运行程序，摄像头的画面就会被显示在ImageMagick窗口中，被识别的物品将会被红色边框标记出来，如图44-1所示。

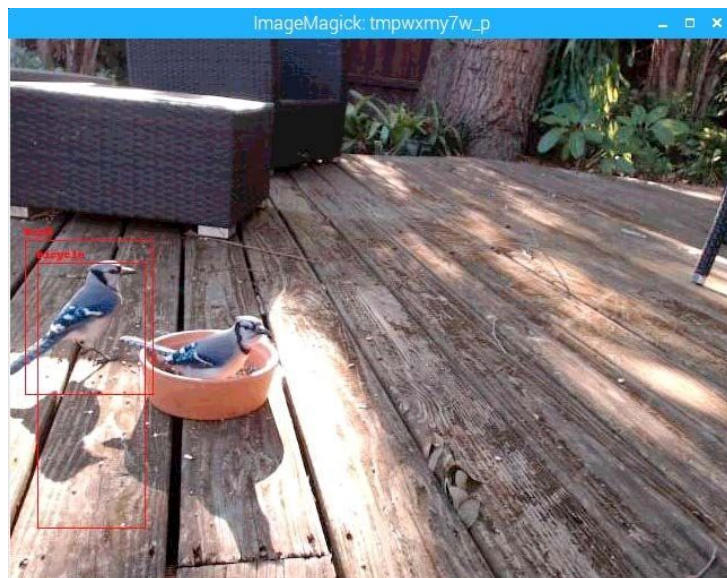


图44-1 图像识别

本章用树莓派和一些简单的脚本^[2]实现了实时物品识别，整个过程看起来很简单，但其中浓缩了很多计算机专家的智慧。通过树莓派这个小硬件，我们让人工智能变得触手可及。

[1] 关于这个安装包的详细信息，可以参考其项目说明<https://github.com/samjabrahams/tensorflow-on-raspberry-pi/>。

[2] 本章中的Python代码参考GitHub代码库<https://github.com/burningion/poor-mans-deep-learning-camera>。

附录A 字符编码

计算机数据本质上是二进制序列。二进制序列可以翻译成一个数字，但不能翻译成字符。在普通人眼里，这些二进制序列不像字符那样可读。如果想在屏幕上显示出一个字符，我们必须知道数据和字符的对应关系。最常见的是将一个字节，即8位的二进制序列对应成英文字符、数字和符号的ASCII编码（American Standard Code for Information Interchange）。

你可以用ascii命令来查询字节和字符的对应。该命令需要通过apt-get安装。命令ascii可以返还一个字节的数字与字符的对应关系，如表A-1所示。

表A-1 ASCII对应关系

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	0	NUL	16	10	DLE	32	20		48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	\$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(56	38	8
9	9	HT	25	19	EM	41	29)	57	39	9
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>
15	0F	SI	31	1F	US	47	2F	/	63	3F	?

续表

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	
77	4D	M	93	5D]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

我们看到ASCII编码中只包含了英文字符，而没有汉字等其他语言的字符。UTF-8是最常见的国际统一编码，编码可以将1到6个字节的二进制序列换成各种各样的字符。在UTF-8的字符集中，有1112064个字符，比128个字符的ASCII强大得多。比如下面符合UTF-8的二进制数据：

1110 0110 1000 1100 1001 0110 1110 0111 1000 0101 1010 0100

根据UTF-8编码对应，实际上是“挖煤”这两个中文字符。

除了ASCII和UTF-8，还有常用于简体中文的GB2312、常用于日文的Shift JIS等编码。值得注意的是，同一个二进制序列，可以根据不同的编码规则翻译成不同的字符文本。根据UTF-8翻译成“挖煤”的二进制序列，可以按照ASCII翻译成：

æ □ □ ç □ ¨

虽然这样的字符文本成立，但是对于阅读者来说没有意义。很多时候，安装的软件出现乱码，或者看到的网页是编码，就是因为选错了字

符编码类型。因此，写入文件和读出文件时，应该选用相同的字符集。

附录B Linux命令速查

这里总结了Linux查询系统下的常用命令。其中，*filename*、*file1*、*file2* 都是文件名。有时文件名有后缀，比如 *file.zip*。

command, 命令名
dir, 文件夹名
string, 字符串
username, 用户名
groupname, 组名
regex, 正则表达式
path, 路径
device, 设备名
partition, 分区名
IP, IP 地址
domain, 域名
ID, 远程用户 ID
host, 主机名, 可以为 IP 地址或者域名
var, 变量名
value, 变量值

1.命令帮助

```
$man command
# 查询命令 command 的说明文档
    $man -k keyword
    # 查询关键字

$info command
# 更加详细的说明文档

$whatis command
# 简要说明
$which command
# command 的 binary 文件所在路径
```

```
$whereis command  
# 在搜索路径中的所有 command
```

这里只是以command (binary file) 为例。比如man还可以用于查询系统函数、配置文件等。

2.用户

```
$finger username  
# 显示用户 username 的信息
```

```
$who  
# 显示当前登录用户  
$who am I  
# 一个有趣的用法
```

```
$write username  
# 向用户发送信息（用 EOF 结束输入）
```

```
$su  
# 成为 root 用户
```

```
$sudo command  
# 以 root 用户身份执行
```

```
$passwd  
# 更改密码
```

3.SHELL (BASH)

```
$history
# 显示在当前 shell 下的命令历史

$alias
# 显示所有的命令别称
    $alias new_command='command'
    # 命令 command 的别称为 new_command

$env
# 显示所有的环境变量
    $export var=value
    # 设置环境变量 var 为 value

$expr 1 + 1

# 计算 1+1
```

4.文件系统

```
$du -sh dir
# 文件夹大小, -h 人类可读的单位, -s 只显示摘要

$find . -name filename
# 从当前路径开始, 向下寻找文件 filename

$locate string
# 寻找包含有 string 的路径
    $updatedb
    # 与 find 不同, locate 并不是实时查找, 你需要更新数据库, 以获得最新信息

$ln -s filename path
# 为文件 filename 在 path 位置创建软链接

$pwd
# 显示当前路径
    $cd path
    # 更改当前工作路径为 path
    $cd -
    # 更改当前路径为之前的路径
```

5.文件

```
$touch filename
```

```
# 如果文件不存在，则创建一个空白文件；如果文件存在，则更新文件读取并修改时间
```

```
$rm filename
```

```
# 删除文件
```

```
$cp file1 file2
```

```
# 复制 file1 为 file2
```

```
$ls -l path
```

```
# 显示文件和文件相关信息
```

```
$mkdir dir
```

```
# 创建 dir 文件夹
```

```
    $mkdir -p path
```

```
    # 递归创建路径 path 上的所有文件夹
```

```
    $rmdir dir
```

```
    # 删除 dir 文件夹，dir 必须为空文件夹
```

```
    $rm -r dir
```

```
    # 删除 dir 文件夹及其包含的所有文件
```

```
$file filename
```

```
# 文件 filename 的类型描述
```

```
$chown username:groupname filename
```

```
# 更改文件的拥有者为 owner，拥有组为 group
```

```
$chmod 755 filename
```

```
# 更改文件的权限为 755: owner r+w+x, group: r+x, others: r+x
```

```
$od -c filename
```

```
# 以 ASCII 字符显示文件
```

6.文件显示

```
$cat filename
# 显示文件
    $cat file1 file2
    # 连接显示 file1 和 file2
```

```
$head -1 filename
# 显示文件第一行
```

```
$tail -5 filename
# 显示文件倒数第五行
```

```
$diff file1 file2
# 显示 file1 和 file2 的差别
```

```
$sort filename
# 对文件中的行排序，并显示
    $sort -f filename
    # 排序时，不考虑大小写
    $sort -u filename
    # 排序，并去掉重复的行
```

```
$uniq filename
# 显示文件 filename 中不重复的行（内容相同，但不相邻的行，不算做重复）
$wc filename
# 统计文件中的字符、词和行数
    $wc -l filename
    # 统计文件中的行数
```

7.文本


```
$echo string  
# 显示 string
```

```
$echo string | cut -c5-7  
# 截取文本的第 5 列到第 7 列
```

```
$echo string | grep regex  
# 显示包含正则表达式 regex 的行
```

```
$echo string | grep -o regex  
# 显示符合正则 regex 的子字符串
```

8.时间与日期

```
$date  
# 当前日期时间  
$date +"%Y-%m-%d_%T"  
# 以 YYYY-MM-DD_HH:MM:SS 的格式显示日期时间（格式可参考$man date）  
$date --date="1999-01-03 05:30:00" 100 days  
# 显示从 1900-01-03 05:30:00 向后 100 天的日期时间
```

```
$sleep 300  
# 休眠 300 秒
```

9.进程

```
$top
# 显示进程信息，并实时更新

$ps
# 显示当前 Shell 下的进程
    $ps -lu username
    # 显示用户 username 的进程
    $ps -ajx
    # 以比较完整的格式显示所有的进程

$kill PID
# 杀死 PID 进程（PID 为 Process ID）
    $kill %job
    # 杀死 job 工作（job 为 job number）
$lsuf -u username
# 用户 username 的进程打开的文件

$dmesg

# 显示系统日志

$time a.out
# 测试 a.out 的运行时间
```

10.硬件

```
$uname -a  
# 显示系统信息
```

```
$df -lh  
# 显示所有硬盘的使用状况
```

```
$mount  
# 显示所有的硬盘分区挂载  
$mount partition path  
# 挂载 partition 到路径 path  
$umount partition  
# 卸载 partition
```

```
$sudo fdisk -l  
# 显示所有的分区  
$sudo fdisk device  
# 为 device (比如/dev/sdc) 创建分区表, 进入后选择 n、p、w  
$sudo mkfs -t ext3 partition  
# 格式化分区 partition (比如/dev/sdc1)
```

修改 */etc/fstab* , 以自动挂载分区。增加行:

```
/dev/sdc1 path(mount point) ext3 defaults 0 0
```

```
$arch  
# 显示架构
```

```
$cat /proc/cpuinfo  
# 显示 CPU 信息
```

```
$cat /proc/meminfo  
# 显示内存信息
```

```
$free  
# 显示内存使用状况
```

```
$pagesize  
# 显示内存 page 大小 (以 KB 为单位)
```

11.网络

```
$ifconfig
# 显示网络接口及相应的 IP 地址。ifconfig 可用于设置网络接口
    $ifup eth0
    # 运行 eth0 接口
    $ifdown eth0
    # 关闭 eth0 接口

$iwconfig
# 显示无线网络接口

$route
# 显示路由表。route 还可以用于修改路由表

$netstat
# 显示当前的网络连接状态

$ping IP
# 发送 ping 包到地址 IP

$tracert IP
# 探测前往地址 IP 的路由路径

$dhclient
# 向 DHCP 主机发送 DHCP 请求，以获得 IP 地址及其他设置信息

$host domain
# DNS 查询，寻找域名 domain 对应的 IP
    $host IP
    # 反向 DNS 查询

$wget url
# 使用 wget 下载 url 指向的资源
    $wget -m url
    # 镜像下载
```

12.SSH登录与文件传输

```
$ssh ID@host  
# ssh 登录远程服务器 host, ID 为用户名
```

```
$sftp ID@host  
# 登录服务器 host, ID 为用户名。
```

sftp登录后，可以使用下面的命令进一步操作。

```
get filename    # 下载文件  
put filename    # 上传文件  
ls              # 列出 host 上当前路径的所有文件  
cd              # 在 host 上更改当前路径  
lls             # 列出本地主机上当前路径的所有文件  
lcd             # 在本地主机更改当前路径
```

```
$scp localpath ID@host:path  
# 将本地 localpath 指向的文件上传到远程主机的 path 路径  
$scp -r ID@site:path localpath  
# 以 ssh 协议，遍历下载 path 路径下的整个文件系统，到本地的 localpath
```

13.压缩与归档

```
$zip file.zip file1 file2
# 将 file1 和 file2 压缩到 file.zip

$unzip file.zip
# 解压缩 file.zip

$gzip -c filename > file.gz
# 将文件 filename 压缩到 file.gz

$gunzip file.gz
# 解压缩 file.gz 文件

$tar -cf file.tar file1 file2
# 创建 tar 归档
    $tar -zcvf file.tar file1 file2
    # 创建 tar 归档, 并压缩
    $tar -xf file.tar
    # 释放 tar 归档
    $tar -zxf file.tar.gz
    # 解压并释放 tar 归档
```

14.打印

```
$lpr filename
# 打印文件

$lpstat
# 显示所有打印机的状态
```

附录C C语言语法摘要

C语言诞生于20世纪70年代初，由贝尔实验室的丹尼斯·里奇与肯·汤普森设计开发。C语言的使用极为广泛，是编译器和操作系统领域最常用的编程语言。这里简要介绍C语言的语法。如果想深入了解C语言，那么请查看附录F中介绍的参考书目，阅读其中和C语言编程有关的书。

1.C语言编写

编写C语言程序，要先用nano文本编辑器来生成以.c为后缀的C源文件，然后按照本书介绍的方式编译运行。

2.注释

注释是代码中的附加文本，用来说明代码功能，从而更好地维护代码，它不会改变代码的运行效果。

- 单行注释：以//开头到该行结尾都是注释。
- 多行注释：以/*开头，以*/结尾，中间内容是注释。

3.函数

C语言主要以函数来组织功能单元。与本书中介绍的bash函数类似，C的函数也用于把多个编程语句组合成一个功能。我们可以把函数理解成一台机器，能接受一些数据作为输入，并返回一个数据作为结果。

```
int sum(int a, int b){  
    int c;          // 声明整数类型变量  
    c = a + b       // 加法和赋值  
    return c;       // 返回值  
}
```

上面的程序定义了一个函数，这个函数的输入是整数a和b，输出是整数c的值。整数c是a和b的和。可以看到，每个语句以;结尾。

4.算术和变量

从上面的程序中，我们已经看到了加法运算和变量。C语言中的算术运算比bash的还要直观，加法、减法、乘法、除法如下：

```
2+4  
b-a  
6/2  
a*7
```

参与运算的可以是数字，也可以是变量。本书介绍了bash语言的变量。C也是用变量在内存中保存数据，但因为C的变量可以有很多类型，比如整数、字符、浮点数等，所以必须先声明变量类型再使用。此外，函数定义的一开始，也必须声明返回值类型。

5.main函数

每个C程序运行时，都会默认运行main函数。

```
int sum(int a, int b){  
    int c;  
    c = a + b  
    return c;  
}  
  
int main(){  
    int a;  
    int b;  
    a = 1;  
    b = 1;  
    c = sum(a, b); // 调用 sum 函数  
    return 0;  
}
```

在上面的main函数中，我们调用了之前定义的sum函数。

6.控制结构

C语言支持像bash那样的选择和循环结构，比如if else的条件选择，又如while和for形式的循环。在用法上，C控制结构也和bash相似。在本书中，你可以看到C语言的例子。

7.函数声明

如果调用函数还没有定义出后面的函数，就需要提前声明函数，从而让编译器没看到函数定义时，就能知道如何使用该函数。例如下面的函数，main函数要调用的sum函数定义在main之后，因此要提前声明函数。

```
int sum(int a, int b);    // 函数声明

int main(){
    int a;
    int b;
    a = 1;
    b = 1;
    c = sum(a, b);
    return 0;
}
int sum(int a, int b){
    int c;
    c = a + b;
    return c;
}
```

如果要调用的函数存在其他程序中，那么也需要声明函数。编写完善的C语言库，会把函数声明包含在一个头文件中。我们只需要在程序一开始引入这个头文件就够了，不需要额外声明函数，比如下面的printf函数。

```
#include <stdio.h>

int main(){
    printf("Hello world!");
}
```

函数printf的声明包含在 *stdio.h* 这个头文件中。

8.指针

C语言存在一种特别的数据类型，即指针。我们知道，变量是内存中的一个位置，而指针就是变量的具体位置。

```
#include <stdio.h>

int main ()
{
    int var;        // 实际变量的声明
    int *p;         // 指针变量的声明

    var = 1;
    p = &var;       // 在指针变量中存储 var 的地址

    // 打印指针中的地址
    printf("Address: %p\n", p );

    // 打印指针指向的数据值
    printf("Value: %d\n", *p );

    return 0;
}
```

根据声明，p是一个变量，可以存储一个指针，该指针只能指向一个整数变量。我们通过&运算符来获取一个变量的地址，而通过*运算符来获得指针指向位置的数据。我们分配堆上的内存空间时，malloc函数返回的就是一个指针。

9.数组

变量只能存储单个数据。C语言提供了数组的语法，可以在连续的内存中存储多个相同类型的数据。比如：

```
#include <stdio.h>

int main(){
    int score[5]={100, 99, 97, 89, 99};
    for(i=0; i<5; i++){
        print("%d", a[i]);
    }
    return 0;
}
```

在声明数组时，要在数组名后面增加[]（方括号）。在上面的声明中，方括号中的5表示数组能容纳的数据总数，用for循环的方式打印出数组中的每个元素。我们可以像使用一个变量那样使用一个元素。在非声明的情况下，数组名后面方括号中的整数表示元素的位置，即数组下标。数组下标从0开始，因此a[1]表示数组a的第二个元素。本质上说，数组名a表示一个指针，保存了数组a第一个元素的内存地址。a[3]实际上相当于*(a+3)。

附录D Makefile基础

当编译一个大型项目时，往往有很多目标文件、库文件、头文件及最终的可执行文件。不同的文件之间存在**依赖关系**（dependency）。比如当使用下面命令编译时：

```
$gcc -c -o test.o test.c
$gcc -o helloworld test.o
```

可执行文件 *helloworld* 依赖于 *test.o* 进行编译，而 *test.o* 依赖于 *test.c*。

UNIX系统下的make工具用于自动记录和处理文件之间的依赖关系。我们不用输入大量的gcc命令，而只要调用make就可以完成整个编译过程。所有的依赖关系都记录在 **Makefile** 文本文件中。我们只要运行make，它会根据依赖关系，自上而下地找到编译该文件所需的所有依赖关系，然后自下而上进行编译。

使用一个示例C语言文件：

```
#include <stdio.h>

/*
 * makefile 演示
 */
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

下面是一个简单的 **Makefile**：

```
# helloworld 是可执行程序
helloworld: test.o
    echo "good"
    gcc -o helloworld test.o

test.o: test.c

gcc -c -o test.o test.c
```

#号起始的行是注释行。观察上面的 *Makefile* 可以发现：

- Target与prerequisite为依赖关系，即**目标文件**（target）依赖于**前提文件**（prerequisite），注意，可以有多个前提文件，前提文件之间要用空格分开。

- 依赖关系后面的<Tab>缩进行是实现依赖关系进行的操作，即正常的UNIX命令。一个依赖关系可以附有多个操作。

用直白的话说就是：

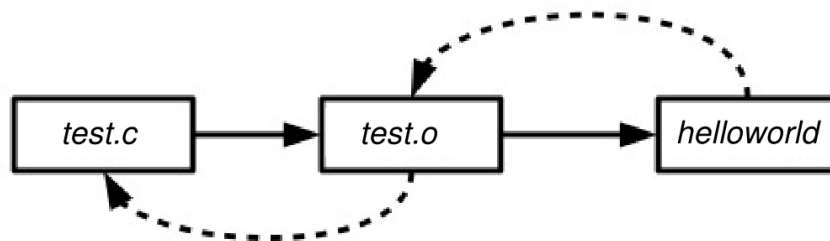
- 想要 *helloworld* 吗？那你必须有 *test.o*，并执行附属的操作。
- 如果没有 *test.o*，那么你必须搜索其他依赖关系，并创建 *test.o*。

我们执行下面的命令来创建 *helloworld*。

```
$make helloworld
```

命令make的执行是一个递归创建的过程，如图D-1所示。

- 如果当前依赖关系中没有说明前提文件，那么直接执行操作。
- 如果当前依赖关系说明了目标文件，而目标文件所需的前提文件已经存在，而且前提文件与上次make时的一样，也直接执行该依赖关系的操作。
- 如果当前目标文件依赖关系所需的前提文件不存在，或者前提文件发生改变，那么以前提文件为新的目标文件，寻找依赖关系，创建目标文件。



图D-1 命令make的执行

在图D-1中，实线表示依赖关系，虚线表示依赖关系检索过程。

上面就是make的核心功能。有了上面的功能，我们可以记录项目中所有的依赖关系和相关操作，并使用make进行编译。

附录E gbd调试C程序

gdb是一款**调试器**（debugger），可用于为C、C++、Objective-C、Java、Fortran等程序debug。在gdb中，你可以通过设置**断点**（break point）来控制程序运行的进度，并查看断点时的变量和函数调用状况，从而发现可能的问题。在许多IDE中，gdb拥有图形化界面。这里主要介绍gdb的命令行使用，并以C程序为例。

1.启动gdb

下面有两个C文件，它们没有bug。我们用gdb来查看程序运行的细节。程序 *test.c* 中有主程序main()， *mean.c* 程序中定义了mean()函数，并在main()中调用。 *test.c* 代码如下：

```
#define ARRAYSIZE 4

float mean(float, float);

int main()
{
    int i;
    float a=4.5;
    float b=5.5;
    float rlt=0.0;

    float array_a[ARRAYSIZE]={1.0, 2.0, 3.0, 4.0};
    float array_b[ARRAYSIZE]={4.0, 3.0, 2.0, 1.0};
    float array_rlt[ARRAYSIZE];

    for(i = 0; i < ARRAYSIZE - 1; i++) {
        array_rlt[i] = mean(array_a[i], array_b[i]);
    }

    rlt = mean(a, b);

    return 0;
```

```
}
```

mean.c 的代码如下：

```
float mean(float a, float b)
{
    return (a + b)/2.0;
}
```

使用gcc同时编译上面两个程序。为了使用gdb对程序进行调试，必须使用-g选项，即在编译时生成debugging信息：

```
$gcc -g -o test test.c mean.c
```

进入gdb，准备调试程序：

```
$gdb test
```

上面的命令是进入gdb的互动命令行。

2.显示程序

我们可以直接显示某一行的程序，比如查看第9行程序：

```
(gdb) list 9
```

显示以第9行为中心，总共10行的程序。我们实际上编译了两个文件，在没有说明的情况下，默认显示主程序文件 *test.c*：


```
4
5  int main()
6  {
7      int i;
8      float a=4.5;
9      float b=5.5;
10     float rlt=0.0;
11
12     float array_a[ARRAYSIZE]={1.0, 2.0, 3.0, 4.0};
13     float array_b[ARRAYSIZE]={4.0, 3.0, 2.0, 1.0};
```

如果要查看 *mean.c* 中的内容，需要说明文件名：

```
(gdb) list mean.c:1
```

可以具体说明所要列出的程序行的范围，比如显示第5到15行的程序。

```
(gdb) list 5, 15
```

显示某个函数，比如：

```
(gdb) list mean
```

3.设置断点

我们可以运行程序：

```
(gdb) run
```

程序正常结束。

运行程序并没有什么有趣的地方。gdb的主要功能在于能让程序在中途暂停。

断点是程序执行中的一个位置。在gdb中，当程序运行到该位置时，程序会暂停，我们可以查看此时的程序状况，比如变量的值。

我们可以在程序的某一行设置断点，比如在 *test.c* 的第16行设置断点。

```
(gdb) break 16
```

你可以查看自己设置的断点：

```
(gdb) info break
```

每个断点有一个识别序号。我们可以根据序号删除某个断点：

```
(gdb) delete 1
```

也可以删除所有断点：

```
(gdb) delete breakpoints
```

4.查看断点

设置断点，并使用run运行程序，程序运行到16行时暂停，gdb显示：

```
Breakpoint 1, main () at test.c:16
16      for(i = 0; i < ARRAYSIZE - 1; i++) {
```

查看断点所在行：

```
(gdb) list
```

查看断点处的某个变量值：

```
(gdb) print a
(gdb) print array_a
```

查看所有的局部变量：

```
(gdb) info local
```

查看此时的栈状态：

```
(gdb) info stack
```

可以更改变量的值：

```
(gdb) set var a=0.0  
(gdb) set var array_a={0.0, 0.0, 1.0, 1.0}
```

当程序继续运行时，将使用更改后的值。

如果我们将断点做如下设置：

```
(gdb) break mean.c:2
```

此时栈中有两个a，一个属于main()，一个属于mean()，可以用function::variable的方式区分：

```
(gdb) print mean::a
```

5.其他

这一部分总结了gdb的其他一些用法。先来说控制程序运行，gdb可以让程序从断点开始，再多运行一行：

```
(gdb) step
```

也可以使用下面命令，从断点恢复运行，直到下一个断点：

```
(gdb) continue
```

使用run重新开始运行。

通过gdb的帮助可以学到更多：

```
(gdb) help
```

或者更具体的命令：

```
(gdb) help info
```

使用下面的命令退出gdb：

```
(gdb) quit
```

附录F 参考书目及简介

Fall, Kevin R., and W. Richard Stevens. *TCP/IP illustrated, volume 1: The protocols*. Addison-Wesley. 2011.

这本书是讲述网络协议的经典著作。这本书篇幅较长，内容较为庞杂，不太适合初学者。可以先阅读Vamei的《协议森林》^[1]等一系列书籍，并积累一定的C语言经验，再来阅读这本书。

Kernighan, Brian W., and Dennis M. Ritchie. *The C programming language*. Prentice Hall PTR. 2006.

Linux主要是由C语言编写的，C语言又是为UNIX而生的语言。良好的C语言基础是Linux开发不可或缺的。这本书是经典的教材，是C语言的创始人写的。虽然它是很薄的一本，但包罗万象。

Membrey, Peter, and David Hows. *Learn Raspberry Pi with Linux*. Apress. 2013.

这本书内容不深，但提供了足够多的细节，在树莓派相关图书里算是写得比较用心的一本了。

Nemeth, Evi, Garth Snyder, and Trent R. Hein. *Linux administration handbook*. Addison-Wesley Professional. 2006.

这本书虽然是手册性质的，但难度不低，适合高级Linux管理员。书中几乎囊括了Linux使用和管理的各种细节，建议至少在阅读本书的前三部分之后再阅读这本书。

Sobell, Mark G., and Matthew Helmke. *A practical guide to Linux commands, editors, and shell programming*. Prentice Hall Professional Technical Reference. 2005.

Linux使用的入门书，介绍基本的命令、Shell编程和编辑器。难度系数较低，适合刚刚接触Linux的用户。

Stevens,W.Richard,and Stephen A.Rago.*Advanced programming in the UNIX environment* .Addison-Wesley.2013.

这本经典书籍对于Linux开发的学习来说有承上启下的作用，也就是如何在Linux kernel和C的基础上写一个合格的Linux应用。书中详细介绍了UNIX的机制及实现方式，同时详细介绍了Linux内核向上层应用所提供的接口。

Tanenbaum,Andrew S.*Modern operating system* .Pearson Education,Inc.2009.

这是一本综合讲述操作系统原理的经典之作，作者也是Minix的作者。Linus就是在该书和Minix的启发下编写出了Linux的，它把内核原理讲得简单易懂。

Torvalds,Linus,and David Read By-Diamond.*Just for fun: The story of an accidental revolutionary* .Harper Audio.2001.

这本书是Linux之父的自传，回忆了Linux操作系统的诞生过程。Linus在里面写了很多八卦，读起来诙谐有趣味。通过他的回忆，我们能更好地理解Linux的编程哲学。

Van der Linden,Peter.*Expert C programming: deep C secrets* .Prentice Hall Professional.1994.

这本书讲述了C语言编程的小窍门。这本书的语言风格通俗易懂，完全没有技术书的枯燥严肃感。作者理解C语言的思路也很精巧，能给人留下深刻印象。

Welsh,Matt,Mathhias Kalle Dalheimer,and Lar Kaufman.*Running Linux* .O'Reilly &Associates,Inc.1999.

它是全面而浓缩的Linux使用指南，就好像一个老手在向你演示如何使用Linux，涉及面较广，但又比较概括，算是一本中级应用指南。

[1] 《协议森林》是一本免费电子书，地址<https://read.douban.com/column/1788114/>。

后记

树莓派正式发布于2012年。相比于一百多年的计算机发展史来说，它是一个年轻的晚辈。从诞生起，树莓派就被学生和硬件开发者追捧。究其原因，它很大程度地降低了人们做硬件创新的成本和难度——一张烧录好操作系统的SD储存卡，配合一小块电路板，就可以组成一个拥有丰富接口的单片机Linux电脑。我们写作这本书，就是想通过树莓派，让大家了解Linux操作系统，了解我们可以通过一个或者多个树莓派来实现有趣的硬件应用。

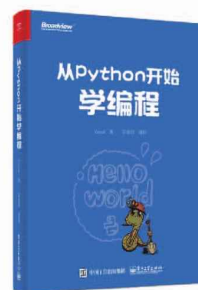
熟悉硬件开发的朋友一定不会对“单片机”这个词感到陌生。单片机在20世纪就已经出现，它是指处理器、存储器、输入输出设备都集成在一个电路板上的微型计算机。最广为人知的“51单片机”就是指所有兼容Intel 8031指令系统的单片机。这些单片机通常被用在嵌入式系统中，作为各种机械、电器设备的控制器。虽然传统的单片机出现很久，但是受限于有限的运算能力，人们必须要编写C语言甚至汇编语言程序才能在单片机上运行。编程技术的高门槛，使得单片机一直只被少数计算机专业人士所使用。

树莓派算是一种单片机，当然人们更多地把它称作系统芯片。相较于传统单片机，树莓派有着强大的运算能力。这使得我们可以让整个Linux操作系统运行在树莓派上。这样的设计，使得我们可以在树莓派上使用任何Linux下常见的编程语言。而树莓派也提供了丰富的输入、输出接口，让我们很容易和其他硬件一起，构造有实用意义的小应用。

我们在写作这本书的过程中切身体会到，技术的进步，让一些原本只能在高校研究院中产生的发明创造，有可能被我们在家中实现。希望读者朋友们通过本书，对Linux和树莓派有更深入的了解，更重要的是，可以自己动手尝试搭建一些应用，让科技服务于自己的生活！

Vamei、周昕梓

好书分享



《从Python开始学编程》
ISBN 978-7-121-30199-5

欢迎投稿：安娜
微信&QQ：80303489
邮箱：anna@phei.com.cn





树莓派开始, 玩转Linux

树莓派的诞生
开始使用树莓派
贝壳里的树莓派
漂洋过海连接你
GPIO的触手
Linux的真身
从程序到进程
万物皆是文本流
会编程的bash
Linux完整架构
函数调用与进程空间
穿越时空的Linux信号
进程的fork
进程间通信
多任务与同步

进程调度
内存分页
Linux分级存储
遍阅网络协议
树莓派平板电脑
天气助手
树莓派博客
访客登记系统
节能照明系统
树莓派挖矿
树莓派高性能计算
蓝牙即时通信
制作一个Shell
初试人工智能



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



策划编辑: 安娜
责任编辑: 汪达文
封面设计: 李玲

上架建议: 网络与互联网

ISBN 978-7-121-34266-0



9 787121 342660 >

定价: 69.00元